AFRL-AFOSR-UK-TR-2011-0040

Innovation Engine for Blog Spaces

Andras Lorincz

Neumann János Számítógép-tudományi Társaság
Eotvos Lorand University
Department of Information Systems
Pazmany Peter setany 1/C
Budapest, Hungary H-1117

EOARD GRANT 07-3077

September 2011

Final Report for 01 September 2007 to 31 March 2011

Air Force Research Laboratory
Air Force Office of Scientific Research
European Office of Aerospace Research and Development
Unit 4515 Box 14, APO AE 09421

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From – To)*
11-10-2011	Final Report	1 September 2007 – 31 March 2011

4. TITLE AND SUBTITLE

Innovation Engine for Blog Spaces

5a. CONTRACT NUMBER

FA8655-07-1-3077

5b. GRANT NUMBER

Grant 07-3077

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Dr. Andras Lorincz

5d. PROJECT NUMBER

5d. TASK NUMBER

5e. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Neumann János Számítógép-tudományi Társaság
Department of Information Systems
Pazmany Peter setany 1/C
Budapest, Hungary H-1117

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

EOARD
Unit 4515 BOX 14
APO AE 09421

10. SPONSOR/MONITOR'S ACRONYM(S)

AFRL/AFOSR/RSW (EOARD)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

AFRL-AFOSR-UK-TR-2011-0040

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The goal of this project was to show, as a kind of Turing Test, how well the machine "understands" ongoing group discussions, the interest of the group, and how well it can participate. There are a number of related problems that we had to solve, including the following: (1) Ethical problem: For proper evaluation, we should not uncover that the participant is a machine. We decided to resolve this by restricting the machine to asking questions about news that could be interesting to a community. For example, by using analogies, datamining can discover that an earthquake has long lasting effects if the road system is poor and can ask about the quality of road system, how good it is and how much it was distorted, and can bring related news for human expert evaluations. (2) Community problem: In order to show the capabilities of present day machine learning techniques and natural language processing methods, we needed a relatively narrow topic domain and a well defined community. (3) Statistical problem: We needed a large a quickly developing database. (4) Problem of contributing: Our original idea, that we would contribute in blogspaces, was inappropriate for contributing: blogspace is not for active discussions. Twitter was suggested, but it has the same problem. These are all passive options, where either somebody's blog is to be commented, or one's own blog is to be created that can gain visibility and reactions. The ideal case that we finally discovered is to contribute and serve to forums. It is, however, harder, since forum texts are highly imprecise, are very short, use slang, topic related TLAs, and unimportant, topic irrelevant text fragments. We found that the number of scientific blogs is small. We decided to move from scientific blogs to blogs on movies, although they are harder, but they solve the community problem. We had to scale up our original crawler architecture to this huge database, collected and analyzed blogs. Now, we learned the problem of contributing, since in this field competition between bloggers is huge. This was the time when we decided to turn to forums. Here, we had to solve the sense disambiguation problem of unknown words. Our progress enables us to contribute to forums. However, we need permission from the owner of the forum. We started negotiations with Sanoma (http://www.sanomamedia.hu/) who are (rightly) asking for proofs on historical data. We transferred our method from English to Hungarian, collected, processed, and evaluated the Hungarian database, including Wikipedia in Hungarian and we are proceeding along this new route. We knew and emphasized in our proposal that testing the technology needs human experts, the collection of devoted experts is impossible or costly, so we have to contribute to human communities. The original idea of contributing to blogspace had several flaws that we had to fix. The present route needs additional work, but it is very promising. We found that there are a number of potential applications, including intelligent portals, intelligent user manuals, everything where the term "frequently asked questions" is appropriate and where context based inference or analogy discovery are necessary or helpful.

15. SUBJECT TERMS

EOARD, Machine Learning, Information Fusion

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18, NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			JAMES LAWTON Ph. D.
UNCLAS	UNCLAS	UNCLAS	SAR	200	19b. TELEPHONE NUMBER *(Include area code)* +44 (0)1895 616187

Innovation Engine for Blog Spaces

Final Report
Contract: AFRL FA8655-07-1-3077

Contact
István Alföldi
John von Neumann Computer Society (NJSzT)

Principal Investigator
András Lőrincz
Eötvös Loránd University

Budapest

September 2011

Contents

Chapter 1

Executive Summary

1.1 Goals, achievements, and potentials

1.1.1 Goals and problems

The goal was/is to show, as a kind of Turing Test, how good the machine "understands" ongoing group discussions, the interest of the group, and how well it can participate. There are a number of related problems that we had to solve.

Ethical problem: For proper evaluation, we should not uncover that the participant is a machine. We decided to resolve this by restricting the machine to asking questions about news that could be interesting to a community. For example, by using analogies, datamining can discover that an earthquake has long lasting effects if the road system is poor and can ask about the quality of road system, how good it is and how much it was distorted, and can bring related news for human expert evaluations.

Community problem: In order to show the capabilities of present day machine learning techniques and natural language processing methods, we needed a relatively narrow topic domain and a well defined community

Statistical problem: We needed a large a quickly developing database.

Problem of contributing: Our original idea that we would contribute in blogspace was inappropriate for contributing: blogspace is not for active discussions. Twitter was suggested, but it has the same problem. These are all passive options, where either somebody's blog is to be commented, or an own blog is to be created that can gain visibility and reactions. The ideal case that we finally discovered is to contribute and serve to forums. It is, however, harder, since forum texts are highly imprecise, are very short, use slang, topic related TLAs, and unimportant, topic irrelevant text fragments.

1.1.2 Achievements

We found that the number of scientific blogs is small. We decided to move from scientific blogs to blogs on movies, although they are harder, but they solve the community problem. We had to scale up our original crawler architecture to this huge database, collected and analyzed blogs. Now, we learned the problem of contributing, since in this field competition between bloggers is huge. This was the time when we decided to turn to forums. Here, we had to solve the sense disambiguation problem of unknown words. Our progress enables us to contribute to forums. However, we need permission from the owner of the forum. We started negotiations with Sanoma (`http://www.sanomamedia.hu/`) who are (rightly) asking for proofs on historical data. We transferred our method from English to Hungarian, collected, processed, and evaluated the Hungarian database, including Wikipedia in Hungarian and we are proceeding along this new route. We are expecting to start a demo next year.

1.1.3 Outlook

We knew and emphasized in our proposal that testing the technology needs human experts, the collection of devoted experts is impossible or costly, so we have to contribute to human communities. The original idea of contributing to blogspace had several flaws that we had to fix. The present route needs additional work, but it is very promising.

We found that there are a number of potential applications, including intelligent portals, intelligent user manuals, everything where the term “frequently asked questions" is appropriate and where context based inference or analogy discovery are necessary or helpful.

1.2 Progress in this semester

In this semester, we have made progress in the following areas:

- **We introduced Structured Sparse Coding to interpret unknown words.** We distinguish two types of unknown words: words whose surface form is corrupted, but the concept they denote is known, and novel words that are completely unknown. Novel words are interpreted as a combination of related concepts by exploiting the distributional hypothesis, according to which words that occur in the same contexts have similar meanings. We introduce regularization that induces structured sparsity in order to exploit this hypothesis and diminish the effect of accidental similarities.

- We negotiated with Sanoma. They are interested in our methods since we – in principle – can support their communities and save those communities from moving to FaceBook. We have moved technology components from English to Hungarian, have accomplished most of the necessary changes,

including the sense disambiguation of unknown words. We are working on quality measures that can prove to Sanoma that the technology will serve their communities.

1.3 Progress in the preceding semesters

In this section, we describe the four principal research directions throughout the project, and the progress achieved in the preceding semesters. The sections that describe detailed results are referenced.

1. **Collecting relevant information:** The main challenge of the project at the beginning was the lack of domain-specific quality corpora. At first we discovered the most relevant blogs in the domain manually, and downloaded them periodically with an automated software tool we developed (Sec. 4.1). This was quickly proven to be inadequate because (i) only a relatively small number of documents could be collected and (ii) not all of the documents were domain-specific. To address these problems, a novel type of web crawler was created (Sec. 4.2) that is capable of reliably collecting millions of domain-specific documents. The system was extended with the capability to identify the definitive websites in a domain based on links (Sec. 4.3), and also on content (Sec. 4.4).

2. **Interpreting text:** Interpreting text fragments has been the main goal of the project from the beginning. Given a suitable, automatically generated representation of text, the other goals, such as novelty detection, and measuring the spreading of information become much easier to reach. Two approaches were proposed:

   (a) **word graphs:** In a word graph $G = (V, E)$, the nodes $v \in V$ are words, and the edges $e \in E$ are relations between the words. We have extracted various relations from text (Sec. 5.1) such as morphological, syntactic and semantic relations. Diffusion was successfully utilized on these graphs to yield a keyword extraction algorithm with better than state-of-the-art performance (Sec. 5.2). Information spreading was detected by finding clusters of similar documents in a corpus (Sec. 5.3). To measure document similarity, various Bag of Words (the cosine, Dice and Jaccard similarity measures) and word graph based (graph kernels) similarity measures were used. For clustering, we used the Clique Finder algorithm, and our own heuristic method. Background knowledge was added by extending the word graphs with synonyms of the word nodes generated from the British National Corpus (Sec. 5.4). The results were promising, however, the algorithms did not scale well to our needs. We turned towards sparse coding and dictionary learning algorithms that, in addition of scaling much better, allowed us to generate more useful and meaningful representations. Our experience with word graphs was utilized in the construction of the crawler and detection of influential blogs.

(b) **topics:** Text fragments can be interpreted as a mixture of topics. The content of each text fragment is represented as a linear combination of topics learned from a corpus. This representation has several advantages over word graphs: (i) it is more intuitive, (ii) the algorithms are much faster and scale better (iii) detecting spreading of information and novelty is straightforward (i.e., two documents are similar if they are described by the same topics; novel documents cannot be described well by existing topics). The topics are mined from a corpus by dictionary learning, and the representation is computed by sparse coding (Sec. 5.5). The topics can be embedded into a topography (Sec. 5.6) where similar topics are close to each other. Interpreting words and text fragments converge nicely in *topography of sense-topics* (Sec. 5.7). In this representation, both the content and the meaning of a single document is interpreted. The topics consist of Wikipedia senses, and each word in the document is interpreted as a combination of these sense-topics. In preliminary experiments, the representation was applied successfully to improve upon Explicit Semantic Analysis.

3. **Interpreting words:** In order to detect information spreading between two text fragments that share only a little content, but a lot of meaning, interpreting the words in them is necessary. In our first attempt, we tried to disambiguate each word to a single Wikipedia sense (Sec. 6.1), but found that the connection between words and senses are not so clear-cut. Next, we took a more flexible approach: each word is interpreted as a *combination of senses*. We used Explicit Semantic Analysis and clustering to obtain sense vectors that characterize the meaning of words. (Sec. 6.2). A more natural approach using Sparse Coding was proposed in the 6th semester (Sec. 6.3), and realized through algorithms based on Subspace Pursuit and Robust Principal Component Analysis in the 7th semester (Sec. 6.3). We found that our framework outperforms state-of-the-art methods.

4. **Scaling up the project:** One of the most important questions throughout the project was that of scale. The algorithms require vast amounts of data (e.g., Wikipedia, domain-specific corpora, etc.). In order to collect this data, we designed a novel crawler. In order to store and access this data efficiently, we designed (Sec. 7.1) and implemented (Sec. 7.2) a new, general purpose software architecture built upon an Information Retrieval system.

Research Direction	Task	Result	Semester
Collecting relevant information	Manually selected blogs	Download a number of manually selected blogs periodically	1st
	Topical web crawling in Blogspace	Collect very large topical corpora from the Web	4th
	Crawl to follow spreading of information	Determine a few websites in a topic to monitor relevant novel content	4th
	Follow spreading of information based on content	An improvement over the previous method	6th
Interpreting Text - word graphs	Extracting word graphs from text	Interpret a text fragment as a graph of words	2nd
	Keyword extraction based on word graphs	Interpret a text fragment as a graph of keywords	3rd
	Document similarity and blog diffusion	Evaluation of several document similarity and clustering algorithms	3rd
	Utilizing synonyms for document similarity	Extend the word graphs with synonyms extracted from a corpus	3rd
Interpreting Text - topics	Learning topics from documents	Interpret a document as a collection of topics	6th
	Learning a topography of topics	Embed the topics into a topography	7th
	Interpreting text fragments	Interpret both the content and the words of a text fragment	7th
Interpreting Words	Translating words to Wikipedia senses	The SenseGraph representation	4th
	Interpreting words as combinations of senses	Research built upon; A new architecture	5th
	Sparse coding to determine meaning	A framework to determine the meaning of unknown words	6th
	Interpreting words	The realization of the framework with different novel algorithms	7th
Scaling up the project	Design of a new architecture	Scale up the project to millions of documents	5th
	Development	Scale up the project to millions of documents	6th

Chapter 2

Interpreting unknown words in context using Structured Sparse Coding

2.1 Introduction

Natural languages are inherently ambiguous. One aspect of this ambiguity is that most words can be used in different *word senses*. A problem commonly encountered when interpreting words in free text is that some of the words we need to interpret are unknown.

The unknown words may be corrupted (e.g., misspelled words, blurred words in scanned documents, errors introduced by Optical Character Recognition or Automatic Speech Recognition), or novel. On the Web, new words are created on a daily basis. The meaning of these new words can seldom be inferred from their surface form (i.e., their written form in the text). Consider the meaning of the word `n00b`. It is nearly impossible to infer: `novice`.

Novel words are unknown in a different way than corrupted words. Although their surface form may not be corrupted, the concept they denote is unknown (e.g., there are no training examples for it). Interpreting a novel word is harder than interpreting a known one with a corrupted surface form.

Novel words may be interpreted as a *combination* of word senses. For example, the word `frape` means "humiliating someone on Facebook who left their profile unattended". Probably, this concept is not present in any sense inventory at the time of writing, but it is possible to represent it as the combination of (`Facebook`, `Humiliation`, ...), where each sense contributes a different aspect.

It is important to note that a word is "novel" when it is novel *to the algorithm*. It is very difficult, if not impossible, to construct and use a sense inventory that contains all the senses we might encounter in free text, even when utilizing rich knowledge sources, such as WordNet or Wikipedia. Sufficient amount of

training examples are even harder to obtain.

According to the *distributional hypothesis*, words that occur in the same contexts tend to have similar meanings [26]. The context of a novel word will be more similar to the contexts of words that denote related concepts than to contexts of unrelated concepts.

Based on these observations, we propose a method to interpret unknown words. The method does not depend on the surface form of the word interpreted, and is capable of interpreting novel words as combinations of known concepts by exploiting the distributional hypothesis.

We interpret a word by approximating its context with the linear combination of *sense-annotated contexts* (i.e., contexts that are annotated with a sense). The interpretation of the word is the vector that contains the coefficients of this linear combination. Each word is interpreted as a combination of a *few* word senses by adding a *structured sparsity* inducing regularization.

As only a few senses can be selected into the interpretation vector, the selected senses will be related to the concept the novel word denotes when the distributional hypothesis is true. One expects that structured sparsity protects against selecting senses that are similar only by accident, as it enforces that only senses that have at least several similar contexts annotated with them are selected.

The method can be generalized, as it can process labeled text fragments as well as sense-annotated contexts. We expect that this more general framework has further applications. In question answering, for example, the text fragments may be definitions labeled with the concept they define, and questions may be interpreted in terms of these definitions to find the answer. Topics could be assigned to documents in a similar way.

We demonstrate the ability of the method to interpret the two type of unknown words on two novel problems. In the first problem, we intend to determine the exact sense of a word whose surface form is unknown. This generalizes the original word sense disambiguation problem since we work with a single, large set of senses instead of a smaller distinct set for each word.

In the second problem, we interpret novel words. This problem is inspired by context clustering, an approach to word sense induction [48]. A complete weighted graph is constructed, where the weights represent semantic similarity, and each node is labeled with a sense. We study the quality of the clustering invoked by the labels.

In the next section we review related work. Our method and results are described in Section 2.3 and 2.4, respectively. We discuss our results in Section 2.5 and conclude in Section 2.6.

2.2 Related Work

To the best of our knowledge, the problem of interpreting corrupted and novel words has not yet been investigated in the literature. However, some similar problems have been addressed in information retrieval and natural language

processing. We review them below.

The problem of corrupted words has been attacked from the perspective of information retrieval. The TREC-5 confusion track [33] studied the impact of data corruption introduced by scanning or OCR errors on retrieval performance. In the subsequent spoken document retrieval tracks [23] the errors were introduced by automatic speech recognition.

Word sense disambiguation is the ability to computationally determine which sense of a word is activated by its use in a particular context (see [48] for a survey). The sense is selected out of a small set of candidate senses for the word, determined by the surface form. Interpreting unknown words is not part of this problem.

These methods can not assign a sense to a word with a corrupted surface form, because the set of senses to choose from is not known. Novel words can not be interpreted, as there is no corresponding word sense in the sense inventory.

We obtain the sense-tagged contexts by mining Wikipedia [44]. Similarly to Mihalcea [45], we think of Wikipedia as a sense-tagged corpus.

Explicit Semantic Analysis [21] assigns weighted vectors of Wikipedia concepts (or senses) to words. An inverted index of Wikipedia is built to map each word to the Wikipedia articles it appears in. The words are not interpreted, as the vector assigned to each word is the same regardless the concept it denotes. The vectors are used to assign a centroid-based interpretation to a text fragment.

Jenatton et.al. [30] demonstrate an application of structured sparsity to a natural language processing task. They apply Sparse Hierarchical Dictionary Learning to learn hierarchies of topics from a corpora of NIPS proceedings papers.

2.3 The method

In this section, we describe the general framework that operates with labeled text fragments. Then we put forth an application of this framework to interpreting words.

Each text fragment is represented in the widely used bag of words representation as a vector $\mathbf{v}$ of weights assigned to words, where v_i is the number of occurrences of the ith word in the text fragment.

The text fragment $\mathbf{x} \in \mathbb{R}^m$ is approximated linearly with the columns of a matrix $\mathbf{D} = [\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_n] \in \mathbb{R}^{m \times n}$, where $\mathbf{D}$ is called the dictionary. The columns of the dictionary contain text fragments. Each column $\mathbf{d}_i$ is labeled with a label $l_i \in L$.

The interpretation vector $\boldsymbol{\alpha}$ is obtained as the coefficients of the linear combination

$$\mathbf{x} = \alpha_1 \mathbf{d}_1 + \alpha_2 \mathbf{d}_2 + \ldots + \alpha_n \mathbf{d}_n. \tag{2.1}$$

$\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_n)^T \in \mathbb{R}^n$ is the interpretation vector assigned to $\mathbf{x}$.

To obtain meaningful representations, it is important to represent each text fragment as a combination of text fragments labeled with only a *few* labels.

Humans prefer to express the meaning of a word, the answer to a question, or the contents of a document as a combination of a *few* concepts. In most natural language processing tasks, a sparse representation is preferred. This sparsity is enforced by introducing a structured sparsity inducing regularization.

The structural constraint is introduced through a family of sets $\mathcal{G} = \{G_l\} \subseteq 2^{\{1,\ldots,n\}}$. There are as many sets in $\mathcal{G}$ as there are distinct labels in L. For each $l \in L$, there is exactly one set $G_l \in \mathcal{G}$, that contains the indices of all the columns $\mathbf{d}_i$ labeled with l. $\mathcal{G}$ forms a partition.

The interpretation vector $\boldsymbol{\alpha}$ for text fragment $\mathbf{x}$ is obtained by minimizing the loss function

$$\min_{\boldsymbol{\alpha}} \frac{1}{2}\|\mathbf{x} - \mathbf{D}\boldsymbol{\alpha}\|_2^2 + \kappa \left[\sum_{l \in L} ||\boldsymbol{\alpha}_{G_l}||_2^{\eta}\right]^{\frac{1}{\eta}} \quad (\kappa > 0), \tag{2.2}$$

where $\mathbf{y}_G$ denotes the vector where all the coordinates that are not in the set $G \subseteq \{1, \ldots, n\}$ are set to zero.

The first term is the approximation error on the observed coordinates, the second is the structured sparsity inducing regularization. Parameter κ controls the tradeoff between the approximation error and the structured sparsity inducing regularization. Parameter η can be set to $0 < \eta < 1$ to achieve a stronger sparsification.

To avoid the overrepresentation of any label, there should be an equal number of columns in $\mathbf{D}$ labeled for label $l \in L$.

The interpretation vector $\boldsymbol{\alpha}$ can be condensed to a single label (e.g., for classification) in two simple steps. First, a new vector $\boldsymbol{\alpha}'$ is created by summing the values in $\boldsymbol{\alpha}$ in each group G_j. To each label $l \in L$, there is a single corresponding coordinate in $\boldsymbol{\alpha}'$, the weight of that label in the representation.

$$\boldsymbol{\alpha}'_l = \sum_{j \in G_l} \boldsymbol{\alpha}_j \tag{2.3}$$

In the second step, the label corresponding to the largest weight in $\boldsymbol{\alpha}'$ is selected.

2.3.1 Interpreting words by exploiting the distributional hypothesis

The method is applied to interpret words by using a dictionary of contexts annotated with senses.

The N-wide context of a word is the N non-stopword words that precede and follow it in the text. A context usually contains $2N$ words, except, for example, the context of a word in the beginning of a document. A stopword is a common function word such as `the`, `a`, `at`, etc.

The columns $\mathbf{d}_1, \mathbf{d}_2, \ldots, \mathbf{d}_n$ of the dictionary $\mathbf{D}$ contain sense-tagged contexts. $\mathbf{d}_i$ is the bag of words representation of an N-wide context of a word, and the label l_i is the sense the word is annotated with. The context of the unknown word goes into $\mathbf{x}$.

According to the distributional hypothesis, words that occur in the same contexts tend to have similar meanings. As the context $\mathbf{x}$ of the unknown word is approximated with the linear combination $\mathbf{D}\boldsymbol{\alpha}$, at least some of the columns $\mathbf{d}_i$ that are selected (i.e. $\boldsymbol{\alpha}_i$ is nonzero) will be similar to $\mathbf{x}$.

Without further constraints, a stronger statement can not be stated. Consider a representation $\boldsymbol{\alpha}$ computed by least squares. In such a dense representation, nearly every coordinate would be nonzero. Additionally, the representation will degrade for exceptions to the distributional hypothesis, as contexts $\mathbf{d}_i$ that are similar to $\mathbf{x}$ only by accident will be selected.

A (non-structured) sparsity inducing regularization ensures that only a few coordinates in $\boldsymbol{\alpha}$ are nonzero. In such a representation, most selected contexts have to be similar to $\mathbf{x}$. But contexts could still be selected based on accidental similarity. Non-structured sparsity is not robust to the potential errors introduced by exceptions to the distributional hypothesis.

The structured sparsity inducing regularization ensures that there will be only a few *word senses* selected. A word sense $l \in L$ is selected if the context $\mathbf{x}$ or part of it can be approximated well linearly in the subspace determined by the columns $\mathbf{d}_i$ labeled with l. The approximation in each subspace can be dense, but only a few subspaces may be selected. Contexts that are similar by accident are not selected, as they are alone in the subspace.

If the meaning of a novel word is closely related to another, known concept, it will be represented fairly accurately, as the known concept will be selected instead. The meaning of words that have no such closely related concepts can be represented as a combination of less related concepts. In the worst case, at least the broad meaning or domain of the word can be determined, provided there are enough word senses in L.

The argument can be extended to many problems in natural language processing, where the semantic similarity of the columns in the dictionary $\mathbf{D}$ and the text fragment $\mathbf{x}$ can be exploited to solve the problem. Consider, for example , centroid-based document classification [25].

2.4 Results

The performance of the method is measured in two problems. The first problem is generalized word sense disambiguation, where the surface form is unknown. In the second problem, novel words are interpreted, whose meaning was never encountered before.

We solve the minimization task of Eq. 2.2 by means of the well established iterated reweighted least squares method. We perform 5 iterations. For the details, please refer to [31].

2.4.1 The datasets

The datasets are obtained by randomly sampling Wikipedia. We consider Wikipedia as a sense-annotated corpus, similarly to Mihalcea [45]. Each

Wikipedia article, identified with its unique title, describes a distinct concept or *word sense*. Each hyperlink is perceived as a sense-annotated word, where the anchor text (i.e., the word) is annotated by the title of the Wikipedia article the link points to (i.e., the word sense). For example, the word `bar` is tagged by the senses `bar (law)`, `bar (counter)`, `bar (establishment)`, etc.

Each dataset consists of sense-annotated contexts $(\mathbf{c}_1, l_1), (\mathbf{c}_2, l_2), \ldots, (\mathbf{c}_C, l_C)$, where the context $\mathbf{c}_i \in R^m$ of an anchor text is annotated with sense $l_i \in L$, the target of the hyperlink.

We do not perform feature selection. However, we remove the words that appear less than five times across all contexts, in order to discard mistyped words.

A dataset is collected from Wikipedia as follows. We download Wikipedia in XML format[1]. We use the English Wikipedia from February 2010. Disambiguation pages, and articles that are too small to be relevant (i.e., have less than 200 non-stopwords in their texts, or less than 20 incoming and 20 outgoing links) are discarded. Inflected words are reduced to their root forms by the Porter Stemming Algorithm [54].

To produce a dataset, a list of anchor texts are generated that match a number of criteria. The anchor text has to be a single word that is between 3 and 20 characters long, must consist of the letters of the English alphabet, must be present in Wikipedia at least a hundred times, and must point to at least two different Wikipedia articles, but not to more than 20. To discard very rare words and abbreviations, the word has to occur at least once in *WordNet* [46] and at least three times in the *British National Corpus* [5].

A number of anchor texts are selected from this list randomly, and their hyperlinked occurrences are collected. An N-wide context of an occurrence constitutes a context $\mathbf{c}_i$ annotated with the title of the article the link points to, l_i.

To ensure that there is an equal number of contexts labeled with each sense $l \in L$, all the contexts that are labeled with a sense that occurs less than d times are discarded. Additionally, contexts are discarded until there is exactly d contexts annotated with each sense.

2.4.2 Word sense disambiguation of words whose surface form is unknown

This problem can be thought of as a generalized word sense disambiguation problem. Given a context $\mathbf{x} \in R^m$ of a word, the sense $l \in L$ the word is used in should be determined without knowing its surface form. The performance of the algorithms is measured as the accuracy of this classification.

The performance of the method is compared to two state-of-the-art methods adapted to this problem. The first is the LIBSVM [10] implementation of a one-against-one multiclass support vector machine (SVM) with a linear kernel. In a study conducted by Lee and Ng, a one-against-all multiclass SVM with a

[1] `http://en.wikipedia.org/wiki/Wikipedia:Download`

linear kernel gave the best results when applied to the traditional Word Sense Disambiguation problem [40]. We use the same parameter setting (C = 1).

According to [28], the one-against-one approach is superior. In some initial experiments using a one-against-all SVM, we arrived at the same conclusion. We only include the latter results. The SVM is trained on the columns of the dictionary $\mathbf{D}$: each column and its label $(\mathbf{d}_i, l_i)$ is a training example.

To assess the improvement of a structured sparsity inducing regularization, the problem is also solved by Subspace Pursuit [15] as sparse coding problem using the same dictionary. The representation $\boldsymbol{\alpha}'$ computed by Subspace Pursuit is condensed to a single sense by a procedure adopted from [64]. For each sense $l \in L$, we compute the residual $\|\mathbf{x} - \mathbf{D}\delta_l(\boldsymbol{\alpha})\|_2$, where $\delta_l : \mathbb{R}^n \to \mathbb{R}^n$ sets all the coefficients in $\boldsymbol{\alpha}$ not labeled with l to zero. The sense l that minimizes this residual is selected.

As the datasets are obtained by randomly sampling Wikipedia, it is important to examine the algorithms on multiple datasets. We generated 10 different datasets, 5 for validation, and 5 for testing. Every experiment is run on five datasets. The mean and standard deviation is reported.

There are $M = 100$ different word senses in each dataset, and $d = 50$ contexts tagged with each sense. The algorithms are evaluated on datasets of different sizes (i.e., d and M are different). These datasets are created by truncating the original datasets (i.e., randomly deleting contexts and their labels).

Stratified d-fold cross-validation is used on each dataset. The dataset is partitioned into d subsets, where each subset contains exactly M contexts - one annotated with each sense. In one iteration, one subset is used for testing, and the other $d - 1$ subsets form the columns of the dictionary $\mathbf{D}$.

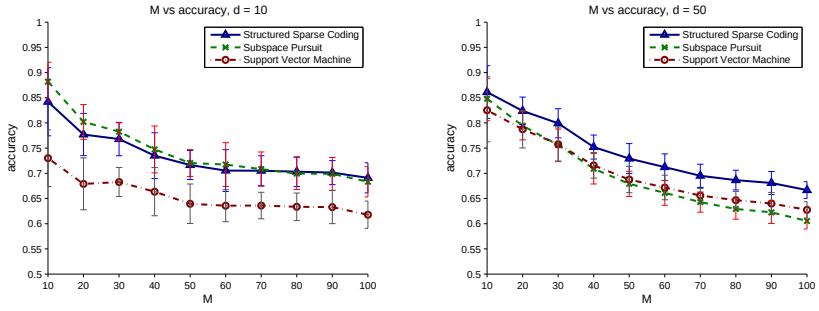
In accord with [40], we use a fairly broad context, $N = 20$. In our own experiments, it was found that a broader context improves the performance of all three algorithms. We set $\eta = 0.5$ to achieve stronger sparsification.

Before evaluating the algorithms, we study how the value of κ (see Eq. 2.2) influences the results. The effect of κ was evaluated on datasets with $d = 20$ and $M \in \{20, 50, 100\}$. Dependence of the accuracy on κ is relatively weak: the mean accuracy changes smoothly between 0.62 and 0.70 (std=0.02) in the range $\kappa \in [2^{-20}, 2^{-1}]$, peaking at around $\kappa = 2^{-8}$ for $d = 20, M = 50$.

As M increased, the optimal value of $\kappa = 2^p$ decreased linearly in its exponent p. The optimal values for $M = 20, 50, 100$ are p = -6, -8, -10, and were the same with $d = 20$ and $d = 50$. We set the value of kappa by linearly interpolating the value of the exponent p between $p = -6$ for $M = 10$ and $p = -10$ for $M = 100$.

The parameter K of Subspace Pursuit was tested using the same procedure. The values $K \in \{5, 10, 20, 30, 40, 50, 80, 100\}$ were tried. We found the performance of the algorithm nearly identical for $40 \leq K \leq 100$. $K = 40$ was chosen, as it gave the best results.

The performance of the algorithms was evaluated on problems with a varied number of distinct senses $M \in \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$ and $d \in \{10, 20, 30, 40, 50\}$. We report the two extremal cases, $d = 10$, and $d = 50$, on Figure 2.1.



(a) (b)

Figure 2.1: Word sense disambiguation of words whose surface form is unknown - results. The accuracy of the three algorithms is compared as the number of senses M grows. Each data point is the mean of five experiments. The error bars denote standard deviations.

2.4.3 Interpreting novel words

The words interpreted in this problem are unknown in two respects: their surface form and the concept they denote is both unknown. This is simulated by taking a number of distinct labels $T \subset L$, and using all the contexts $\mathbf{c}_i$ labeled with some $l \in T$ as test examples. The contexts labeled with some $l \in L \setminus T$ constitute the dictionary $\mathbf{D}$. The labels in the dictionary and the labels of the test examples are disjoint.

The evaluation procedure is inspired by word sense induction. Word sense induction algorithms try to automatically identify the set of senses a word can be used in in an unsupervised manner [48]. One of the main approaches in word sense induction is context clustering. The contexts a word occurs in are clustered into groups, each identifying a sense of the word.

As the labels $l \in L$ already determine the clusters, the performance of the algorithms can be measured as the *quality* of this clustering.

A simple, complete graph $G = (V, E, \omega)$ with edge weighting $\omega : E \to \mathbb{R}_+$ is constructed, where each node $v \in V$ is a test example labeled with a label $l \in T$. The edge weighting ω is determined by computing the *semantic similarity* between the representation vectors of each pair of nodes.

The vectors could be compared using a variety of similarity measures [65]. We use cosine similarity, as it is the most widely known and used. Negative similarities are set to zero. It is computed as

$$sim(\mathbf{a}, \mathbf{b}) = \frac{< \mathbf{a}, \mathbf{b} >}{||\mathbf{a}||_2 ||\mathbf{b}||_2}. \tag{2.4}$$

A partition $\mathcal{C} = \{C_l\}$ of V is defined by the labels $l \in T$. For each distinct label $l \in T$ there is exactly one cluster C_l that contains the nodes labeled with l.

	Representation		
Senses	Bag of Words	Subspace Pursuit	Structured S. C.
50	0.330 ± 0.054	0.339 ± 0.093	$\mathbf{0.354} \pm 0.043$
100	0.336 ± 0.055	0.410 ± 0.106	$\mathbf{0.451} \pm 0.060$
200	0.341 ± 0.056	0.475 ± 0.1	$\mathbf{0.519} \pm 0.048$

Table 2.1: Interpreting novel words - results. As the number of senses $|L \setminus T|$ that represent the meaning of the novel word grows, the sparse coding based interpretation vectors perform significantly better than the bag of words representations of the contexts.

There are many *quality indices* to evaluate whether a given clustering is of high quality [55]. We opt to use *coverage* [7], one of the simplest measures. Let the weight of all intracluster edges (i.e., the edges whose endpoints are in the same cluster) be denoted by $\omega(\mathcal{C})$, and the weight of intercluster edges (i.e., the edges whose endpoints are in different clusters) by $\overline{\omega}(\mathcal{C})$. Coverage measures the fraction of the weight of intracluster edges with respect to the total weight of all edges:

$$coverage(\mathcal{C}) = \frac{\omega(\mathcal{C})}{\omega(\mathcal{C}) + \overline{\omega}(\mathcal{C})}. \tag{2.5}$$

Intuitively, the larger the value of $coverage(\mathcal{C})$, the better the quality of the clustering $\mathcal{C}$. A mincut has maximum coverage. Usually, a mincut is not considered optimal, as in many cases it separates an individual vertex from the rest of the graph. However, as the clusters in $\mathcal{C}$ are fixed and of the same size, coverage can be considered a good measure.

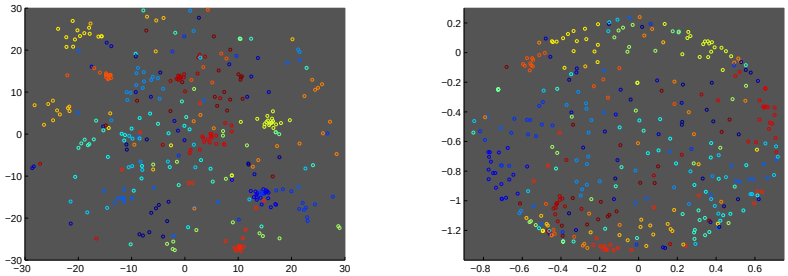
There are a number of qualities a clustering of high quality should possess that are not reflected in $coverage(\mathcal{C})$. Additional measures could be introduced (e.g., intracluster conductance), but we chose a different, more straightforward approach. To better see the structure of the graph G, it is embedded into two dimensions with two low dimensional embedding algorithms: Multidimensional Scaling [37] and t-Distributed Stochastic Neighbor Embedding [59].

Three datasets are used to examine the effect the number of senses in $L \setminus T$ has on the coverage. Each dataset has $d = 20$ contexts per sense, and $|L \setminus T| = 50, 100, 200$. Five sets of missing senses T are selected, with $|T| = 20$. Each experiment is conducted on these five sets.

The method is compared to two baselines: the bag of words representation of the contexts $\mathbf{c}_i$, and the representation $\boldsymbol{\alpha}'$ computed by Subspace Pursuit.

The following parameters were set to the same values as in the previous problem: $N = 20$, $\eta = 0.5$, $K = 40$. Dependence of the coverage on κ is strong: the mean coverage changes between 0.24 and 0.32 (std=0.04) in the range $\kappa \in [2^{-4}, 2^{-8}]$ peaking at around $\kappa = 2^{-6}$ for $|L \setminus T| = 50$.

Parameter $\kappa = 2^{-6}$ was set for $|L \setminus T| = 50, 100$, and $\kappa = 2^{-7}$ for $|L \setminus T| = 200$. In this problem, the optimal κ does not depend as strongly on M as in the first



(a) t-Stochastic Neighbor Embedding (b) Multidimensional Scaling

Figure 2.2: The graph G generated using the Structured Sparse Coding representation embedded into two dimensions. The nodes in the same cluster are marked by the same color.

problem.

Table 2.1 reports the coverage values obtained by the three representations on the three datasets. Figure 2.2 shows the graph G generated by the Structured Sparse Coding representation embedded into two dimensions. The nodes cluster according to the senses they are labeled with.

2.5 Discussion

We demonstrated the ability of the method to interpret the two types of unknown words in two problems. In the first problem, the methods based on sparse coding significantly outperform the Support Vector Machine when there are only a few training examples. Structured sparse coding has the best performance when the size of the dictionary is large. Subspace Pursuit falls behind, possibly because it selects concepts that are accidentally similar to $\mathbf{x}$ (see Section 2.3.1).

In the second problem, when the similarity scores are computed based on interpretation vectors, the nodes naturally cluster according to the senses they are labeled with, even though the concepts they denote are unknown to the algorithm. Both methods based on sparse coding outperform the raw bag of word representation of the concepts.

Structured sparse coding performs the best. As the number of senses to represent meaning of the novel word grows, the methods perform significantly better. The results were obtained with a randomly selected set of senses L. In real-world applications, better results could be obtained by adjusting L to fit the problem.

To better understand how novel words are interpreted, it is helpful to take

a look at some interpretation vectors. Table 2.2 shows four good (i.e., chosen specifically for demonstration) interpretation vectors assigned to four concepts. In the first interpretation vector, a broader concept, `Number` was selected instead of the unavailable `Prime number`. The second interpretation vector contains closely related senses to the concept. The third vector is more interesting: each selected sense describes a different aspect of the concept. `Transformers` are `Humanoid` robots (`Cyborg`) that can change into vehicles (`Tram`), and they are also `Heroes` that appear in comic books (`Flash (comics)`) and animated series.

Prime number	Existence	Transformers (toy line)	Departments of France
Number	Logos	Humanoid	Duchy of Burgundy
	Karma	Tram	Burgundy (region)
	Eternity	Flash (comics)	Count
	Mushroom	Cyborg	Farm
		Hero	

Table 2.2: Some good interpretation vectors of novel words. Each column in the table contains an interpretation vector. The first row shows the sense the interpreted novel word was annotated with. The rows below contain the word senses that were selected (i.e. were nonzero in the interpretation vector $\boldsymbol{\alpha}$). The senses are ordered according to their aggregated (i.e., summed) weight in descending order from top to bottom.

2.6 Conclusion

We have proposed a method based on structured sparse coding to interpret unknown words. We distinguish two types of unknown words: words whose surface form is corrupted, but the concept they denote is known, and novel words that are completely unknown. Novel words are interpreted as a combination of related concepts by exploiting the distributional hypothesis. The structured sparsity inducing regularization protects against the accidental similarity of exceptions to the distributional hypothesis.

The experimental results show that the method performs well when there are only a small number of examples available, and it can interpret novel words even as a combination of randomly selected concepts. A generalized method was described that works with labeled text fragments. We expect that this generalized method has further applications in natural language processing.

Chapter 3

Recommending relevant information to forum users

3.1 Goals

We are currently working with *Sanoma Budapest*[1], a leading media company in Hungary and daughter of *Sanoma*, which is one of the largest Eastern European media companies. In this cooperation they provide relevant data for us. If we succeed on those data, then finally we will have a true testing method, and the major bottleneck of our project will be overcome. However, the first step was to redo everything in Hungarian.

Our aim is to recommend relevant news articles to forum users, based on the context of their activity. This context is represented as a piece of text, and can mean the contents of the forum they are currently browsing, or of the post they are currently editing; so the task is to retrieve the most relevant news articles for a given query text.

This feature would help users by supplying them with information, and would also generate more traffic for the news pages. A wide range of methods are used to accomplish this goal.

The news articles we use are from the website *hír24*[2] (*hír* is the Hungarian word for news). This website is a news portal with articles about a wide range of topics, like politics, fashion and technology. For our experiments, we used a collection of roughly 200,000 Hungarian news articles from hír24 covering a year long timespan.

[1] `http://www.sanomamedia.hu/sanobp_english/`
[2] `http://www.hir24.hu`

3.2 Adaptation of our software to Hungarian

In our previous work, we worked only with English texts. To work with Hungarian texts, a number of changes had to be made in our software:

- We had to correctly handle the character encoding, because in Hungarian, there are multiple letters which contain accents, and different character encodings represent these letters differently.
- We had to replace the Porter stemming algorithm with the Hungarian Snowball stemmer[3], to handle the complicated inflections present in Hungarian language.
- We had to change our list of English stopwords to a Hungarian one.
- For experiments with Explicit Semantic Analysis, we had to modify our ESA software because of the differences of the Hungarian Wikipedia to its English counterpart. For example, the names of the templates which denote disambiguation pages are different in the two languages. The Hungarian Wikipedia was downloaded, and then processed and indexed with our modified ESA software.

We indexed the news article collection using *Apache Lucene*. The date of the creation of the article was also indexed as a numeric field. We used a simple BOW-based index searcher as a baseline algorithm.

3.3 Online Structured Dictionary Learning

An OSDL-based (Sec. 5.5.1) system was implemented and used to discover the latent topics present in a news collection with n documents, and to determine the *topics* of single articles.

We use the word "topic" here in a similar sense as in a generative model, but we do not use a probabilistic interpretation: we consider a topic to be a vector which assigns weights to words. These topics are learned to describe articles: we want the tfidf-vectors of the articles to be reconstructed by a linear combination of a small number of topics.

The main steps of this process are the following:

1. A lexicon was compiled from the k most frequent words in the articles not present in our stopwords list. For most experiments, we used the $k = 10,000$ most frequent non-stopwords.
2. The articles were preprocessed for OSDL. A *tfidf*-vector was generated from each article using the lexicon, and statistics from the collection. Each vector was normalized according to l_2 norm independently. A $k \times n$ sized matrix X was built from these tfidf-vectors as columns.

[3] `http://snowball.tartarus.org/algorithms/hungarian/stemmer.html`

3. OSDL was used to factorize the matrix as $X = DA$, where D is a matrix of size $k \times d$, and A is a matrix of size $d \times n$. The D matrix is the dictionary of topics: each of the d columns corresponds to a topic. We denote the i^{th} column of A as $\boldsymbol{\alpha}_i$. Thus, $\boldsymbol{\alpha}_i$ is the description of the i^{th} article as the combination of some topics. In our experiments, the topographies were embedded into hexagonal grids on a torus of various sizes.

4. The components of each $\boldsymbol{\alpha}_i$ were thresholded, discretized, and indexed along with the corresponding article.

Steps 1. and 2. are straightforward, but the later steps involve some parameters that must be tuned correctly for the process to produce good results.

Two parameters that are closely related and very important are the number of topics, and κ, the approximation-regularization tradeoff. We tested with topographies of $20 \times 20 = 400$, $25 \times 25 = 625$ and $32 \times 32 = 1024$ topics. We found that a higher number of topics gives more accurate results, but the training time (the time required to learn the D matrix) grows very sharply when we increase the number of topics, which renders more than 1024 topics infeasible.

The other important parameter, κ must be chosen carefully. When κ is too low, overfitting occurs, and the matrix A will not be sparse enough: too many topics can be active at the same time, which makes the topics to be obscure, and unusable for information retrieval. When κ is too high, the error of the reconstruction becomes too large: the topics do not describe the articles properly, and the topics also become obscure. For different numbers of topics, a different κ is optimal. Figure 3.1. shows some of the effects of changing the κ parameter.

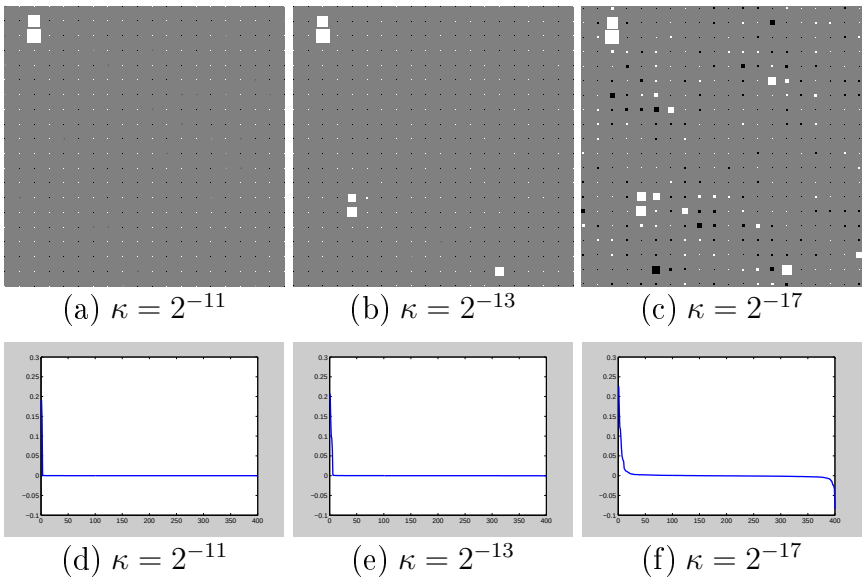
Based on experimentation, we chose κ to be 2^{-13}, 2^{-15} and 2^{-17} for 400, 625 and 1024 topics, respectively. This way, we got a very good topic structure with reasonable topics. For example, these are the five most influential words of a topic (translated to English here): *level, start, firefighter, fire, apartment*[4], which are clearly words related to fire accidents. Another topic: *infection, disease, vaccine, epidemic, virus*[5]. Figure 3.2. shows some topics on this topography.

The vectors $\boldsymbol{\alpha}$ are *compressible*: they are sparse except for noise (i.e., components with very small but nonzero values). We filter out this noise by thresholding. We discard each component with absolute vale below some fixed threshold θ. To determine the value of θ, we generated a histogram of the values in a sample of 2000 $\boldsymbol{\alpha}$ vectors (Figure 3.3.). The histogram shows an exponential distribution on the components with weight above 0.05, so the weights below 0.05 are considered as noise. We used $\theta = 0.05$ as threshold.

The weights in the $\boldsymbol{\alpha}$ vectors are continuous values, to be indexed in a Lucene text field. These fields contain word-frequency pairs, and the frequency value is an integer, so the weights had to be discretized. We followed a simple approach

[4] In Hungarian: emelet, keletkezett, tuzoltó, tuz, lakás

[5] In Hungarian: fertozés, megbetegedés, oltás, járvány, vírus



(a) $\kappa = 2^{-11}$ (b) $\kappa = 2^{-13}$ (c) $\kappa = 2^{-17}$

(d) $\kappa = 2^{-11}$ (e) $\kappa = 2^{-13}$ (f) $\kappa = 2^{-17}$

Figure 3.1: **The effect of parameter κ on the computed $\boldsymbol{\alpha}$.** The figures show differently computed $\boldsymbol{\alpha}$-vectors of the same text, on the same D matrix, with different κ parameters. The Hinton-diagrams on the top show the values of $\boldsymbol{\alpha}$ represented in a 20×20 topography. The graphs on the bottom show the values of $\boldsymbol{\alpha}$ sorted in decreasing order. $\kappa = 2^{-17}$ is too small, the weights of the topics are not sparse enough, and are noisy. $\kappa = 2^{-11}$ provides few active topics. When this parameter is used during OSDL, the topics themselves are not intelligible, and are not useful. $\kappa = 2^{-13}$ seems to be the optimal parameter. For different topographies, a different κ is optimal.

and divided the weights into three bins, with 0.095 and 0.175 as limits. This discretization ensured largely equal frequencies in all three bins.

3.4 Robust Principal Component Analysis

RPCA (Sec. 6.4.7) decomposes a matrix M as $M = L + S$ where L is a low-rank matrix and S is a sparse matrix containing noise. Applying RPCA on a matrix, and using only the low-rank term instead of the original matrix can be used to filter out noise, which makes it an easier task for another algorithm (e.g., OSDL) to learn a dictionary. We think that OSDL might greatly benefit from noise filtering, because the presence of a topic does not guarantee that all the words of the topic occur in the text; and also the occurence of a word does not guarantee the presence of the topic where the word belongs.

To improve our results, we experimented with RPCA to reduce the amount of noise present and also to reduce the dimensionality of the data. This can

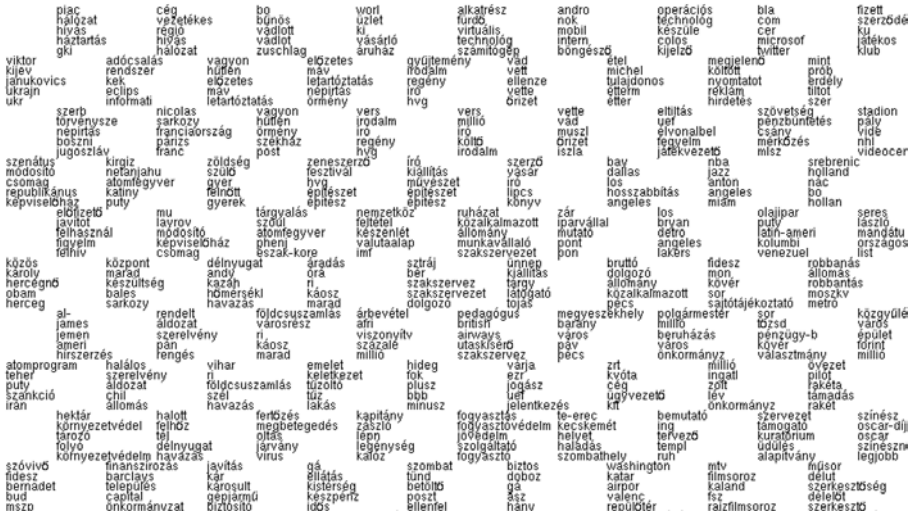


Figure 3.2: **Topography of topics in Hungarian.** This is part of a topography that OSDL generated from the news collection. Words in a single topic tend to be strongly related. See the text of Sec. 3.3 for the English translations of some of the words here.

be achieved by inserting the following operations after step 2 of the process described in Sec. 3.3.

1. The matrix X is multiplied by an $r \times k$ sized R matrix of random values from a standard normal distribution. The new RX matrix is now dense, which is beneficial for RPCA.

2. RPCA is used to decompose the RX matrix to a sparse term of outliers and a low-ranked matrix.

3. SVD is used on the low-ranked term to determine the new low-dimensional space, and to project the low-ranked matrix L to that space. In the following steps, the resulting matrix is used instead of X.

The results in our preliminary experiments were inconclusive. In some cases the results were improved upon, but sometimes it had a negative effect. Further investigation is needed with RPCA to examine its benefits in our system.

3.5 Query construction and scoring

In *Lucene*, documents can have multiple *fields*. A field stores and indexes data about a document that belong together. For example, a document may hold a field for its title, another field for its URL, and another one for the actual

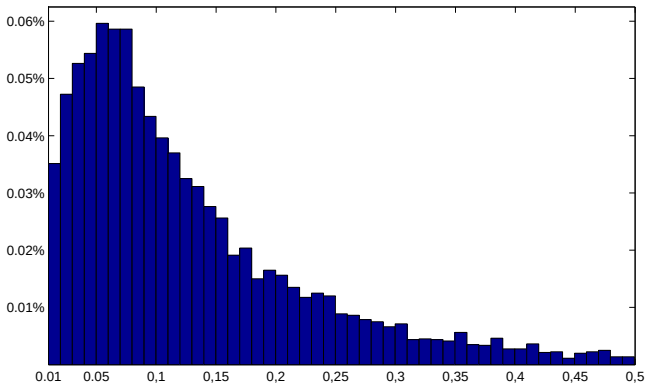


Figure 3.3: **Histogram of topic weights in the $\boldsymbol{\alpha}$ vectors.** The figure shows a histogram of the absolute values of every weight in a sample of 2000 $\boldsymbol{\alpha}$ vectors. The histogram shows an approximately exponential distribution above weight 0.05. Note that the bar below weight 0.01 is not shown because the frequency of weights below 0.01 is vastly above the others.

contents of the article. We stored many fields for the hír24 news articles, of which 3 is used during searching and scoring the documents:

- **Body**: it contains the stemmed words of the article, along with their frequencies (the number of times they occur in the article).
- **Topics**: it contains the OSDL topics of the aricle with weights greater than a specific threshold. The topics are stored as words, therefore they also have frequencies, which is their discretized weight.
- **Date**: the date of the creation of the article, stored as a numeric field.

During searching, the following are the main steps:

1. At first the system generates a tfidf-vector from the query text with the same lexicon and document frequency statistics as was used with the news articles. This vector is normalized according to l_2 norm.
2. Then, the $\boldsymbol{\alpha}$-computing step of the OSDL algorithm is run on the vector, using the D matrix which contains all the topics that OSDL learned before the indexing. The resulting $\boldsymbol{\alpha}$ vector is the description of the query text as the combination of some topics from D.
3. The components of the $\boldsymbol{\alpha}$ are thresholded and discretized with the same parameters as during the indexing.

To search on the text of the articles, we select the most important words of the query text (based on their tfidf-values), and construct a Lucene query from them. Searching based on the text body alone is our baseline algorithm. This algorithm has the drawback that it does not find articles that share the same topic as the query text, but contain different words. Searching on the topics alone would not distinguish between articles that share the same topic, but are about slightly different things. To eliminate these individual shortcomings, the two approach is combined: the search itself is done on two fields (body and topics).

Lucene provides a number of ways to construct sophisticated queries that search multiple fields. Subqueries can either be connected as MUST clauses (then only those articles are returned which contain the word or topic that is in the query) or SHOULD clauses (then the specified word or topic does not have to occur in the document, but if does, the document gets a higher score). We would like to find the most relevant articles even if they do not match our criteria exactly, so we used SHOULD clauses, and found that this approach works well.

The system we descibed so far finds relevant articles, but an additional step should be made to take into account the date of the article. When recommending news articles, it is especially important to prefer newer articles. We implemented a `CustomScoreQuery` class for Lucene which weights the original scores according to date. Currently, it gives double score to articles that are less than a day old, and gives standard score to articles that are a year old or older, with linear interpolation between the two dates. Of course, as the final version of the system will work with real-time news, and not a fixed archive of articles, a more refined approach may be neccessary. For example, another point could be added to the date scoring function at the point of one week before the search.

We made steps to incorporate the reconstruction error of the individual OSDL topic ($\boldsymbol{\alpha}$) computations ($||x - D\boldsymbol{\alpha}||_2$) into the score of the hit. The problem was that, contrary to our expectations, this error had too little bearing on the quality of the topics found. Many times, the algorithm found topics that we found good and useful, but the reconstruction error was high. The error was very high overall, and varied around 0.85. This indicates that a different measure of error should be used.

3.6 Splitting the collection into news categories

OSDL has a parameter that sets the number of topics generated. After some experimentation, we found that increasing the number of topics improves the results, but increasing it above 32×32 was not feasible due to the very high computational requirements. It was desirable that we partition the collection in a meaningful, non-overlapping way. The other thing we found that there are some very obvious false positive hits which needed to be eliminated.

For these reasons, we decided to treat articles that belong to different news *categories* (e.g., economy, technology, entertainment, world news, etc.) sepa-

1.	18.7%	domestic
2.	15.9%	world
3.	13.6%	celebrities
4.	13.3%	science, technology, IT, cars
5.	11.3%	economy, finance
6.	9.4%	accidents, crimes
7.	7.4%	design, fashion
8.	7.8%	entertainment, film, music
9.	2.6%	funny

Table 3.1: **Distribution of categories in the news collection.** The table shows the nine new category groups we used to split the news collection into smaller parts. The second column shows the percentage of the news articles falling into each category.

rately. The authors of the articles classified each article into exactly one category. There were a total of 39 categories, but many of them contained very few articles, and there were very similar ones (such as the categories technology and science). We dropped the smallest categories, and defined 9 equivalence classes among the remaining ones. Table 3.1. shows the classes, and their size. From now on, we use the word *category* for these equivalence classes. We created, processed and indexed 9 non-overlapping collections of news articles, each from a different news category. For each category, we conducted a separate OSDL run with 32×32 topics, so we had a total of 9216 topics.

Of course, splitting the collection along categories introduces another problem, namely, that the system needs an additional input with the query: the category in which it has to search. In forums, it does not pose a big problem, because most forum threads have a clearly defined topic, so the category need to be set only when a new topic is created. Moreover, most forums have a hierarchical structure of topics, which makes the situation even better: children of a forum topic tend to share their category with their parent.

Nevertheless, we also experimented with the automatic categorization of articles into topics, because a fully automatic system would be much more useful, and would also work in applications where there is no strictly defined forum topic. We trained linear SVM classifiers on the tfidf-vectors of the articles. We used linear SVM for its simplicity, speed and also its good performance in text classification tasks. We measured 75% accuracy using 10-fold crossvalidation,.

In later experiments, we augmented the vectors with additional features: an OSDL $\boldsymbol{\alpha}$-computation was run for each article, for each category (regardless of the true category of the article). The matrices D were learned beforehand without using the test articles. 78% accuracy was reached. We also tried using the reconstruction errors of OSDL computations, but the results did not increase considerably. Overall, we expected a better results. The SVM-based automatic classification of forums is an even more challenging task, therefore we concluded

that more advanced methods are needed to automatically categorize them.

3.7 Explicit Semantic Analysis

We found *Explicit Semantic Analysis* (Sec. 6.1.1) to be very useful in characterizing a word or text fragment previously. Using ESA concept vectors instead of simple word tfidf vectors has the advantage of a higher dimensionality. Semantic relatedness measures can be improved by augmenting the tfidf vectors of the texts that need to be compared [22].

Though using articles only from the correct category eliminates the most obvious false positive hits, some can still occur. We experimented with ESA to filter the hits. For this, we indexed the Hungarian Wikipedia, and built an ESA interpreter using the index.

During searching, a similarity value is computed for each hit, based on the ESA vectors of the article and the query text. False hits tend to have clearly lower similarities to the query text than truly relevant articles, but it is also clear that a simple global threshold is not enough to correctly filter out false positive hits, because the suitable threshold varies from query to query.

We plan more experiments to find a method that is efficient and works well for all queries in reducing the number of false hits.

3.8 Implementation

We used MATLAB for the massive numerical computations on collections of tfidf-vectors prior to the indexing phase: for OSDL and RPCA.

We used Java for almost all other operations: for processing the news collection and generating tfidf-vectors from them; for indexing textual and OSDL topic information and searching on them; for building ESA vectors and filtering hits based on them; and for automatic categorization of queries. For the software to function without MATLAB once the indexing is done, we reimplemented in Java the parts of the OSDL and RPCA algorithms that are required to searching. We developed a Java searching server that receives query requests through a socket connection, searches the index, and sends the relevant articles back through the socket.

We augmented this server application using the *Apache JCS*[6] (Java Caching System) library to cache the results of searches, and a number of different partial results: the topics of tfidf-vectors, the ESA-vectors of articles, etc.

We implemented a plugin for the *Invision Board* portal engine in PHP which augments the forum interface with a searching option. This plugin can be used to test our system. The PHP plugin acts like an interface, and uses the Java-based searching server through a socket connection to recommend relevant news articles based on the query text.

[6] `http://commons.apache.org/jcs/`

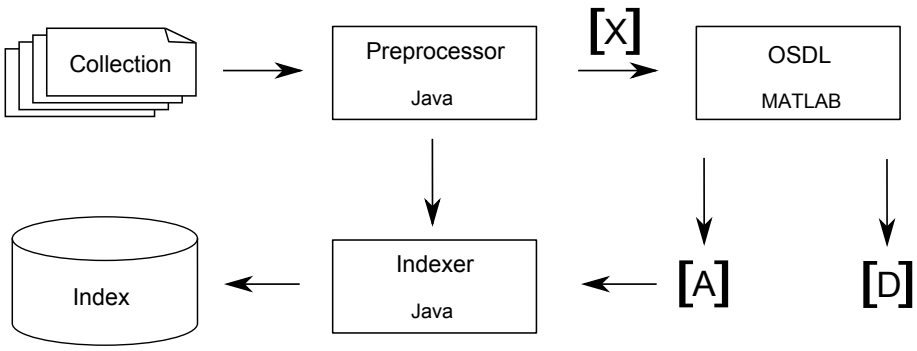


Figure 3.4: **The indexing architecture.** The preprocessor component (written in Java) represents the news collection as the X matrix, which is given to the OSDL algorithm (written in MATLAB). The produced D matrix is used during searching. The indexer creates a Lucene index from the preprocessed articles and from the topics given in the A matrix.

3.9 Results

Testing of the system was done by quierying it using real-world forum entries, and assessing the results based on human judgements. This first step does not give us a precise number, only impressions. In the second step, standard information retrieval datasets (e.g., TREC datasets) will be used to test our system.

Using the baseline algorithm (i.e., searching only for the words that appear in the query text) is very effective in itself, but only when the language of the query text is similar to the language of the articles. In the case of our application (i.e., recommending news articles to forum users) this works if the users on the forum use the language of journalists, which is usually not the case. For example, the user writing to a forum about environmentalist issues might be interested in a science-related news article about the loss of biodiversity, even if he or she does not use the exact words *biodiversity* or *environmentalism*, and instead uses the words *green*, *kill* and *animal*.

A related case where the baseline algorithm might fail is where the news article is about a closely related, but slightly different topic than the query text. It might happen that the two texts do not share a single important word, but are still closely related. For example, a user browsing articles about computer hard drives might be interested in an article about graphics cards as well.

We use OSDL to handle this problem. Of course, for this to work without the use of an external corpora or thesaurus, the words denoting the same topic have to occur together in *some* articles. Our system is often able to cope with situations like that, when there is enough data in the collection about the words, and the situation is not very ambigous.

However, the system is far from perfect. There are often cases when each

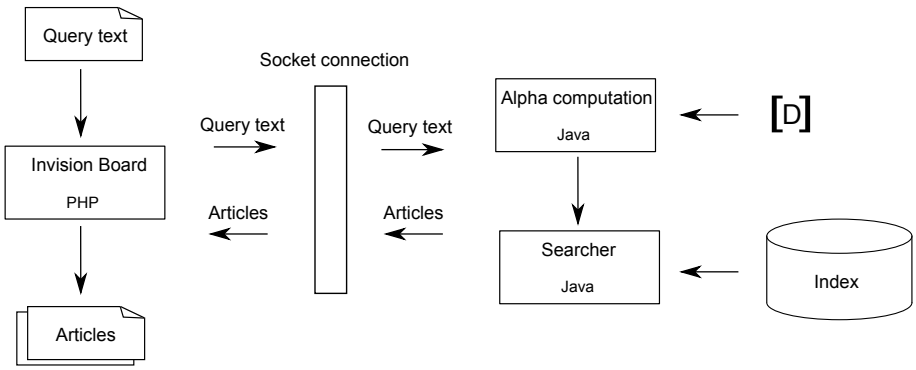


Figure 3.5: **The searching architecture.** Our plugin in the Invision Board forum engine (written in PHP) takes a query text and sends it to the server through a socket connection. The $\boldsymbol{\alpha}$-computing algorithm in it (which is a part of OSDL) discovers the topics in the query text using the D matrix created during indexing. The searcher uses the index to retrieve relevant documents using the words and topics of the query text, and sends the results back to the forum engine, which presents them to the user.

version gives equally bad results. Sometimes the augmented searcher gives worse results. For example, numbers may give a separate topic in OSDL. A query text that contains many numbers may promote that topic, so the results are articles that contain lottery numbers. Overall, we can safely say that the OSDL-augmented version clearly outperforms the baseline as a whole.

3.10 Future Plans

We have many plans to improve our system. Currently, we think that enforcing a topography on the topics gives better results, but we did not test this hypothesis thoroughly. We also did not use the topography explicitly in the system, though this seems feasible: topics that are close to each other tend to be similar.

Overall, the system looks promising: our judgement is that it clearly gives superior results compared to the simple BOW-based baseline, but further tests on standard information retrieval datasets are required to verify this. We are currently planning to test our system using the TREC datasets. We also would like to compare the use of OSDL to similar, standard information retrieval algorithms, such as Latent Semantic Indexing and Latent Dirichlet Allocation.

Chapter 4

Progress in collecting relevant information

4.1 Manually selected blogs

As we planned, we selected a number of blogs for experimentation in the upcoming semester. The topic of most of these blogs is Artificial Intelligence, which enables us to observe and validate the results also manually (besides creating quantitative measures of performance as well), since we have competence in the field. We collected the following blogs:

- `http://www.hunch.net/`: a blog created by John Langford for academic research on machine learning and learning theory. Other researchers also send posts.
- `http://www.illigal.uiuc.edu/web/blog/`: Blog of the Illinois Genetic Algorithms Laboratory (IlliGAL). IlliGAL studies evolutionary and genetic algorithms.
- `http://herselfsai.com/`: A blog dealing with artificial intelligence and brain research.
- `http://medal.cs.umsl.edu/blog/`: Blog of the Missouri Estimation of Distribution Algorithms Laboratory (MEDAL) on evolutionary computing and machine learning.
- `http://geneticargonaut.blogspot.com/`: A blog run by Marcelo de Brito about artificial intelligence, neural networks and their application.
- `http://www.inma.ucl.ac.be/ francois/blog/index.php`: A blog run by Damien Francois dedicated for machine learning and artificial intelligence. Its purpose is tho enumerate the latest techniques and devices for the average user.

- `http://www.kurzweilai.net/`: Ray Kurzweil's blog in scientific topics including artificial intelligence.
- `http://scienceblogs.com/`: a collection of blogs in topics of science such as artificial intelligence.
- `http://blogs.nature.com/wp/nascent/`: Nature's blog on web technology and science.
- `http://www.realnanotechinvestor.com/`: A blog on nanotechnology.
- `http://nanotechwire.com/`: Weblog on the industrial applications of nanotechnology and scientific research in the field.
- `http://www.cnn.com/`: CNN's weblog on general news. Included as a basis for comparisons.

The entries in the blogs contain the date of appearance, which enables us to track topic spreading.

We developed software to update our database of documents coming from these blogs using RSS/Atom and crawlers, which continuously check and download new content. RSS and Atom is an XML based web syndication format for information disclosure implemented by web feeds. Basically, it allows programs to download content in an easy way, in a structured format. Crawlers are pieces of software that periodically check the selected blogs searching for new entries, download the web pages and extract relevant text content throwing away unnecessary parts such as advertisements and HTML formatting tags.

4.1.1 Wikipedia for experimentation

As an alternative for blogs, we also considered running experiments on Wikipedia. The basic idea is that entries in Wikipedia are similar to blog entries, and it might be useful for testing algorithms designed for measuring document similarity. For easier experimentation, we downloaded Wikipedia to obtain a static version of January 3, 2008. A version containing changes as well would also be available, however, it is infeasible for us because of its enormous size (approximately several thousands of gigabytes).

4.2 Topical web crawling in Blogspace

One of the subgoals of the Blogspace project is topical web crawling in Blogspace. It is topical crawling, because its goal is to find documents or web pages that belong to a particular topic. And the crawler operates in Blogspace, that is, our domain of interest is the domain of web logs, or blogs.

So far our topic of interest was chiefly science blogs and news, but as we progressed with the project, we found that there may be not enough blogs whose topic is scientific research. Another problem with science is that there

are not many people that understand it well, so the target audience for our human-computer dialog system would be severely constrained. In light of these developments, we decided to choose another domain that is more accessible to the layman and more popular. We think that a perfect candidate would be blogs about movies, because of the following reasons:

- They are perfectly accessible to anyone, no special knowledge is required to discuss them.
- There are a lot of blogs about movies.
- It is easy to decide whether an article is about a movie or not; movies is a well defined domain.
- Many people are interested in movies, the target audience of our human-computer dialog system is large.
- The truth of a statement about movies can not be determined as easily in the movie domain as in the science domain. This means that the generation of believable content is easier in the movie domain.

We have three goals we can solve by an effective topical crawler:

1. Find as many instances of a document on the blogosphere as possible
2. Find the most influential blogs about a topic
3. Find many relevant documents belonging to a topic

The first two tasks are necessary for the measurement of information diffusion. If we have a piece of information (a document), we would like to know how influential this piece of information turned out to be. Particularly, we would like to know how many blogs posted about our documents (this is the first task) and how influential these blogs are (the second task). The second task also makes the first task easier, because we can start our search for the posts about our documents from the most influential blogs.

In the third task the crawler collects a corpus of topical documents for our human-computer dialog system.

We developed a crawler architecture to solve all three tasks. The architecture is based on Heritrix, the Internet Archive's open-source, extensible, web-scale, archival-quality web crawler. A concise summary of Heritrix can be found in [47]. We extended the Heritrix architecture with various modules in order to implement our focused crawler. From now on we set aside implementation details, and describe our crawler at a higher level.

The novel ideas in our crawler are the following. Firstly, we capitalize on the fact that most blogs are about a single topic. In other words, web pages on a blog constitute a dense cluster of documents about the same topic. The task of our crawler is to find as many such documents clusters as possible. So the basic unit of our crawl is the blog. In contrast, traditional focused crawlers

make their decisions at the lower webpage level. We maintain a list of blogs that can be considered for crawling, where there is a relevance score for all the blogs. The higher the relevance score of a blog is, the more frequently we download pages from that blog.

Secondly, we consider our crawl as a process in time. During a crawl, there are periods when there are many relevant blogs waiting to be downloaded, and there are periods with only a small number of relevant blogs. If we can detect these periods, then it makes sense to slow down our crawl when there are only a few relevant blogs, and speed it up when there are many. The reason for this is the assumption that relevant blogs link to other relevant blogs, and irrelevant blogs link to irrelevant blogs. So we slow down our crawl in order to avoid crawling irrelevant blogs.

The following section is about our crawler architecture and crawling experiments.

4.2.1 The crawler architecture

The basic unit in the architecture is a blog. We identify a blog with its unique domain name. Usually the domain name is the address of the main page of a blog, from which the posts can be accessed. We compose a list of the possible domain names from `http://weblogs.com`, as described in the previous section. Note that although there are blogs without unique domain names, in that case we consider all blogs under a unique domain name as a single unit, because

- important blogs usually have a unique domain name
- if blogs share a domain name, they are usually about the same topic

At the center of the architecture is a list of *active blogs*, that is, the blogs we currently crawl. In this list there are (*domain name*, *relevance rating*) pairs. The *domain name* identifies the blog, and the *relevance rating* represents how close that blog is to our topic. Specifically, it is computed as the estimation of the probability that a document retrieved from that blog will be relevant

$$relevance\_rating_{blog} = \frac{relevant_{blog}}{all_{blog}} \tag{4.1}$$

where $relevant_{blog}$ denotes the number of relevant documents retrieved, and all_{blog} denotes all the documents retrieved from a blog. We decide whether a document is relevant using a *support vector classifier* (see Sect. 4.2.4).

Web pages from blogs that have higher relevance rating are downloaded more frequently. A separate queue belongs to each blog that contains the URLs waiting to be downloaded from that blog. There is a wait time associated with each queue (or blog): the time to wait between two downloads from that queue. This *wait time* depends upon the *relevance rating* of the blog and a parameter (see below).

At first, we initialize the list of *active blogs* using a list of seed blogs. We only download pages from the list of active blogs, and we expand this list based on the assumption that relevant blogs link to other relevant blogs. If we encounter a new page that is not part of an active blog, we note that it was linked from our *active blogs*, and count the number of times it was linked. If this count exceeds a *link count threshold*, we put this blog into the list of *active blogs*, and start to download it.

Figure 4.1: **The representation of one blog** Blogs are represented with a complex data structure. The blog is identified by its *domain name*. The *relevance rating* shows how relevant this blog is to the topic we are interested in. The crawler retrieves web pages from this blog every *wait time* (in milliseconds). The *URL queue* contains all the URLs we know that have not been downloaded yet from the blog.

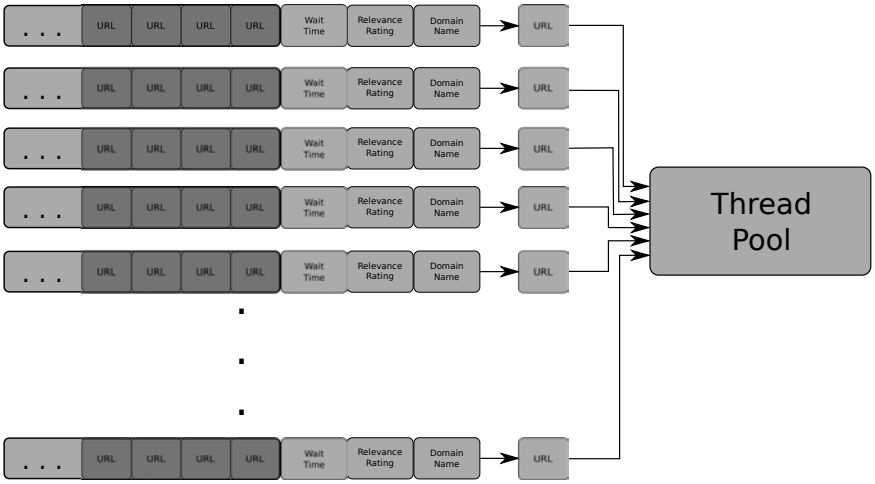


Figure 4.2: **The crawler architecture** The basic units in the architecture are blogs. There is a *wait time* and an *URL queue* associated to each blog. When *wait time* milliseconds have elapsed since the last retrieval from the blog, an URL is taken out of the *URL queue*, and handed over to the *Thread Pool*. A thread in the *Thread Pool* takes the URL, downloads and processes it. Meanwhile, a new *wait time* is computed depending on the *relevance rating* of the blog.

4.2.2 The wait time as a function of the relevance rating

To define the *wait time* between two downloads from the same blog as a function of the *relevance rating* of the blog, we introduce a simple mathematical model. First, we formalize the concept of relevance and relevance rating. For every blog $B \in blogs$ there is a probability $P(d \in R | d \in B)$ that document d retrieved from blog B will be in the set of relevant documents, R. We can estimate this probability for each $B \in blogs$ by the *relevance rating*:

$$relevance\_rating_B = \frac{|\hat{R_B}|}{|\hat{D_B}|}, \tag{4.2}$$

where $|\hat{R_B}|$ is the number of relevant documents encountered so far on this particular blog $B \in blogs$, and $|\hat{D_B}|$ is the total number of documents encountered on this blog.

Our basic assumption is that if only a small percent of the documents on a blog were relevant so far, then it is unlikely that we will find more in the future. The reason is that blogs are usually highly thematic. For example, a blog that contains movie reviews is unlikely to contain articles about kitchen utensils. There are also non-thematic blogs, for example when people blog about their everyday life, but for the purposes of thematic information gathering these are mostly irrelevant.

Based on this assumption, we want to penalize blogs that have a small *relevance rating* much more than blogs with average or high *relevance rating*. As we compute the *wait time* for each blog independently, in the following we will talk about a particular blog $B \in blogs$, and denote its relevance rating $relevance\_rating_B$ by the symbol p. We define the wait time to be

$$f(p) = p^{-\gamma} - 1, \tag{4.3}$$

where γ is a parameter.

The *wait time* function $f(p)$ satisfies all the following requirements:

- It does not penalize blogs with nearly maximal *relevance rating* as $f(p)$ is near 0 when p is near 1, and penalizes smaller p-s more than larger ones. The strength of this penalization can be controlled with the parameter γ.
- As the *relevance rating* gets smaller, $f(p)$ penalizes the blog more and more. The steepness of $f(p)$ is increasing drastically as we go from $p = 1$ to $p = 0$. This means that we stop downloading irrelevant blogs quickly.
- As the *relevance rating* gets larger, $f(p)$ penalizes it less and less. The steepness of $f(p)$ is decreasing drastically as we go from $p = 0$ to $p = 1$. Blogs with higher *relevance rating* will be sampled more frequently, and the *relevance rating* of relevant blogs increases with sampling. We will download relevant blogs even if a considerable fraction of the first few pages retrieved from them were irrelevant.

- The γ parameter enables flow control. If the number of relevant blogs is small, then we have to permit less relevant blogs to continue downloading. Conversely, if we can choose from a large number of relevant blogs, then we can permit only highly relevant blogs to download.

There are a few practical considerations in computing the *wait time function*. The first is that to estimate $P(d \in R | d \in B)$, we have to have enough samples. So if we encounter a new blog, we do not apply any wait time to it until we have enough samples to estimate p. The number of samples to collect is also a parameter.

We also have to define a maximum wait time T in order to deal with the case when $p = 0$. The wait time $f(p)$ can never be more than T. If $f(p) > T$, or $p = 0$ (i.e., $p^{-\gamma} = \infty$), then we set $f(p) = T$.

4.2.3 Flow control: setting the γ parameter dynamically

In this section we introduce our method used to control the flow to satisfy some constraints of the crawl by means of the the γ parameter. During a crawl, we would like to maximize the *relevance rating* (i.e., the percentage of downloaded documents that are relevant), and minimize the *wait time* between URLs (i.e., the average time elapsed between retrieving two documents)[1]. These are two conflicting goals.

When there are many relevant pages, we can decrease the *wait time* freely, but when there are but a few, we have to increase it, or the *relevance rating* will drop, and the crawler will go to irrelevant territory. When there are enough new relevant documents, we can decrease the *wait time* once again.

We can control the balance of *relevance rating* and *wait time* by changing the γ parameter. By increasing γ, the preference for relevant blogs will increase with respect to the irrelevant blogs, and the *wait time* will increase among all blogs. On the other hand, if we decrease γ, then the crawl will speed up, but the ratio between the downloading frequency of relevant and not so relevant blogs will decrease, so *relevance rating* slips.

Minimizing solely the *wait time* one would create a crawler that is not focused at all: it would download the same number of documents from each blog. Maximizing only the *relevance rating* one could easily stop the crawl: the *wait time* would increase so much that crawling and downloading would practically stop.

As a first step towards balancing these two attributes of the crawl one can specify a minimum *wait time*. If the crawl slows down so *wait time* falls below this value we decrease γ until it is above this minimum.

Now that we keep our crawler from going too slow, we can treat the maximization of the *relevance rating*. For this, we want to control the *harvest rate*, a traditional measure of crawler performance. The *harvest rate* is the ratio of the

[1] Note that these concepts correspond to the concepts of the previous section, except that now we consider *all* blogs.

relevant collected pages to all the collected pages in a window of n downloaded web pages.

We introduce two new parameters: one for the minimum target harvest rate, and one for the maximum. The minimum works alike to the *wait time*: if *harvest rate* falls below this threshold, γ is increased, so more relevant pages will be downloaded.

The maximum wait time is necessary, because no blog can have a 100% harvest rate. First, classification of blog pages is never perfect. There will always be pages that are misclassified as irrelevant. Second, there are usually auxiliary pages on blogs that are not relevant but necessary for navigation. Third, *harvest rate* is an aggregate score between all blogs, and some blogs will be less relevant than others. Thus, we specify the maximum so that we do not seek for *relevance ratings* higher than this maximum. This means that we consider higher relevance rates most unlikely, we will not try to go above this value, since it could slow down our crawler considerably.

Wait time and *harvest rate* are estimated as follows. We use the last n document downloads. The *wait time* is calculated as a moving average of the last n wait times between downloading individual documents. The *harvest rate* is calculated in a similar manner; it is the number of relevant documents downloaded over the last n downloads divided by n (the number of all the downloaded documents).

4.2.4 Classifying the blog pages

To classify blog pages we use a *support vector machine* (SVM) classifier [60]. It was shown by Joachims [32] that SVMs are efficient at determining the topic of documents. In particular, we use a linear support vector machine with the dual coordinate descent method and L_2 loss function described in [27]. Our SVM is a binary classifier that decides whether the document belongs to the topic we are interested in.

Generating the feature vectors

In order to generate the feature vectors for the support vector machine, we have to do preprocessing on the raw HTML documents.

First, we extract the textual content of the HTML documents, that is, the text of the main article without the clutter the HTML file contains (e.g., tags, javascript, menu items on the page, etc.). In order to do that, we parse the HTML file into the standard *document object model* (DOM) format using the open source tool HtmlCleaner (`http://htmlcleaner.sourceforge.net/`).

Then we run a simple heuristics on the DOM tree of the HTML file that extracts the main text. It works as follows. The algorithm looks for text at DOM nodes while traversing the DOM tree. It adds the text to the extracted textual content if and only if it is longer than a predetermined constant (100), and the number of alphanumeric characters is at least 80% of all the characters in the text. In other words, if the text is long enough to be part of an article (menus

are short, for example), and if it contains enough alphanumeric characters to be a text, it is added to the extracted article text. Experiments have shown that this simple heuristics is adequate for our purposes.

The next step is stemming the extracted textual content, that is, removing case and inflection information from words. For this task we use the Porter Stemmer, the de facto standard stemming algorithm in natural language processing [54].

After stemming we create a bag of words (BoW) representation (`http://en.wikipedia.org/wiki/Bag_of_words_model`) from the stemmed textual content. In the BoW representation the order of the words in the document is lost, we keep only the frequency of distinct words. We count the number of occurrences for each word separately, and create an attribute-value representation of the text (w_i, n_i), where w_i is the ith distinct word and n_i is the number of occurrences of that word in the text.

We generate the feature vectors from training documents considering the whole collection of documents. First, we establish a unique index for each word that occurs in the document collection. A feature vector $\mathbf{f}$ will contain as many elements as the number of unique words in the document collection. The ith element f_i of the feature vector contains the corresponding entry in the bag of words representation of the document, that is, the number of occurrences of the word associated with the index i in the document.

During crawling we do not have a document collection as we decide about the documents separately. However, we can use the indices obtained from the document collection we trained the SVM on. We map the words of the document onto these indices. If a word can not be mapped onto an index, it will be discarded. As we have a large number of training documents, the number of discarded words will be minimal.

Training the support vector machine

We used five corpora for the training of the SVM. The positive examples were taken from the IMDB's archive for the rec.arts.movies.reviews newsgroup (`http://www.imdb.com/Reviews/`) that contains only movie reviews. Choosing the negative examples was somewhat harder, because we had to incorporate examples from all conceivable topics we might encounter on the web. We included four different corpora into our set of negative examples:

1. The Reuters-21578 corpus, widely used for text classification tasks, contains mainly articles dealing with finance and economics.

   `http://www.daviddlewis.com/resources/testcollections/reuters21578/`

2. The 4 Universities Data Set contains WWW-pages collected from computer science departments of various universities.

   `http://www.cs.cmu.edu/afs/cs/project/theo-20/www/data/`

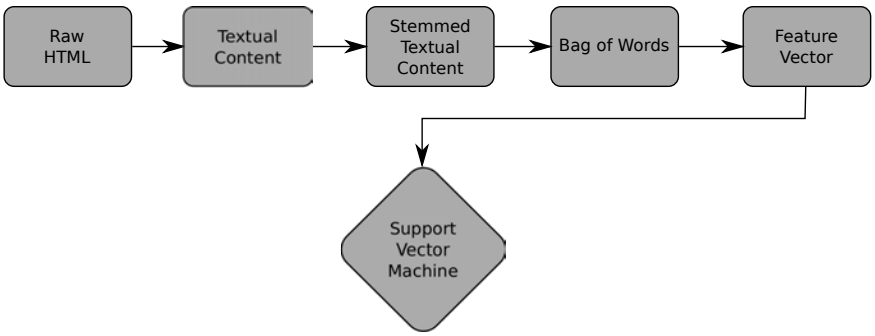


Figure 4.3: **The process of classifying documents** Documents are preprocessed in various ways. First, the textual content is extracted, then it is stemmed. A bag of words (BoW) representation is created from the stemmed textual content. From the BoW representation feature vectors are generated, where the dimension of the feature vector equals to the number of different words in the corpus. The feature vectors represent the documents for the support vector machine (SVM). The SVM is trained to decide whether a document belongs to the topic we are interested by means of its feature vector.

3. The 20 Newsgroups Data Set is a collection of approximately 20,000 newsgroup documents, partitioned (nearly) evenly across 20 different newsgroups. This is a diverse collection of different topics.

   `http://people.csail.mit.edu/jrennie/20Newsgroups/`

4. The Industry Sector Data Set consists of corporate web pages classified into a topic hierarchy with about 70 leaves.

   `http://www.cs.umass.edu/~mccallum/data/sector.tar.gz`

Altogether we had 28000 positive and 56000 negative examples. We conducted two initial experiments before integrating the classifier into our crawler. First, we conducted a 10-fold cross-validation on the $28000 + 56000 = 84000$ training documents, and measured *precision* and *recall*, where

$$precision = \frac{correctly\ classified\ as\ positive}{all\ classified\ as\ positive} \tag{4.4}$$

$$recall = \frac{correctly\ classified\ as\ positive}{all\ positive\ documents} \tag{4.5}$$

Here *positive* means belonging to the topic we are interested in.

Our results are convincing: SVMs are good for such classification. Our *precision* is 0.993 and our *recall* is 0.998, giving rise to an F value of 0.995.

We also downloaded movie reviews and other articles from various sites to test the effectiveness of our classifier on data completely unrelated to our training examples. We found that our classifier could decide whether a page was a movie review or not with nearly 100% accuracy.

4.2.5 Identifying blogs

We would like to constrain our web crawler to the Blogspace, that is, we only want to crawl blogs. So, in order to decide whether to consider a web page for crawling, we have to classify it as a blog page (i.e., part of a blog) or a non-blog page. We do so with the help of blog ping servers.

Every time a blog is updated, it can notify one of these ping servers. The largest and most popular ping server is `http://weblogs.com`. The URLs of the updated blogs can be downloaded from this ping server. The number of URLs that can be collected is enormous: the list we used in our crawler experiments consists of all the unique blog domains that were updated in August, 11 million unique blog URLs. It is safe to assume that most of the interesting blogs are in this list, as blog search engines operate based on these lists.

But not all of these blogs contain useful information: we have to consider spam blogs, or splogs. According to [36], a splog is simply a web spam page that is a blog, with web spam page defined as

> Any deliberate human action meant to trigger an unjustifiably favorable relevance or importance of some page, considering the page's true value.

There are several methods for blog spam filtering, we tried the combination of a few simple heuristics that showed to be quite effective in [36]. We selected the two heuristics that have very high precision and high recall: the presence of a *generator* meta tag (precision: 1.0, recall: 0.75) and the presence of link to an RSS/Atom feed (precision: 0.96, recall: 0.90). A blog is classified as not a spam blog if either the meta tag or the link is present on its main page.

We conducted crawling experiments with and without spam blog filtering, and found that our focused crawling was sufficient to filter out spam blogs, because on spam blogs there were less relevant documents than on legitimate blogs. Upon experimenting, we decided not to use spam filtering to prevent any possibility that we filter out legitimate relevant blogs.

4.2.6 Experiments: collecting topical blog entries

The aim of this experiment was to evaluate our crawler's capability to collect topical blog entries. The traditional measure of focused crawler performance is the *harvest rate* [9]. The *harvest rate* is the ratio of the relevant collected pages to all the collected pages in a window of n downloaded web pages. We chose $n = 100$.

We also monitor the *cumulative harvest rate*, which is the ratio of all the relevant pages collected up to a point (i.e., a number of pages downloaded) to all the pages collected up to that point.

In addition to the harvest rate, we are also interested in the efficiency of our crawl. Because we are to use it as a corpus collection tool in our human computer dialog system, and also for determining the most influential blogs in

a topic, it has to be able to collect large amounts of blog entries in a relatively short time.

After a few test crawls, we set the parameters of our crawler as follows. We set the *link count threshold* to 200: we start to download a new blog if the blogs already being downloaded link to it at least 200 times. First we experimented with smaller values, but the number of blogs downloaded simultaneously increased dramatically. In addition, because we have to download from each blog a sufficient number of entries in order to have a reasonable estimate of the relevance of that blog, the *harvest rate* also decreased. We found 200 to be large enough that not too many pages from irrelevant blogs will be downloaded for estimation, and small enough, i.e., the number of relevant blogs downloaded is not too small.

For the same reason (i.e., that if we download a large number of entries for estimation, the *harvest rate* decreases), we set the *minimum number of pages downloaded for estimation* to 10, a fairly small number. We found that it is large enough, because estimation is getting better and better for relevant blogs as we download more and more blog entries.

We tried to achieve the maximum *harvest rate*, so we set unattainable *minimal and maximal targeted harvest rates* for our crawler. The values were 0.8 and 0.9 for the minimal and for the maximal targeted harvest rate, respectively. We also set the *maximal wait time* such that our crawler could maintain an average download speed of 4 entry/sec. The number of recent downloads to compute moving averaging (i.e., n) was set to 100. We used a constant amplitude update for γ; in each step it was ± 0.01, depending on the sign of the updating. For all blogs, the maximum wait time was set to 8 hours.

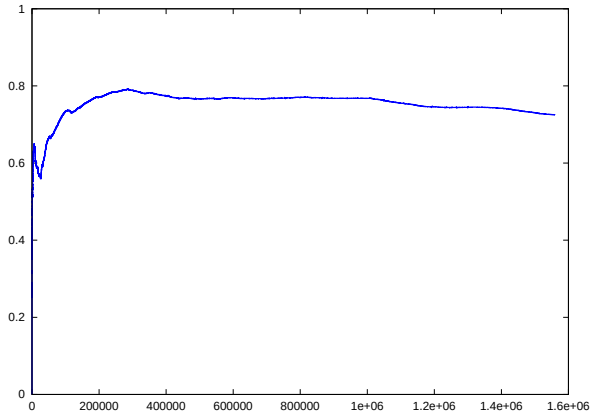
We initialized our crawler with seed blogs from the 'Best of the Web' blog directory. Particularly, we used the blogs in the Arts/Movies/Reviews category (`http://blogs.botw.org/Arts/Movies/Reviews/`). The starting seed list included only 17 blogs. In the course of the crawl, our crawler found more than 4000 blogs starting from these 17.

The list of possible blogs was collected from `http://weblogs.com`. We collected all the blog domains that were refreshed in August. This yielded 11 million unique blog domains, and defined the search space of our crawler.

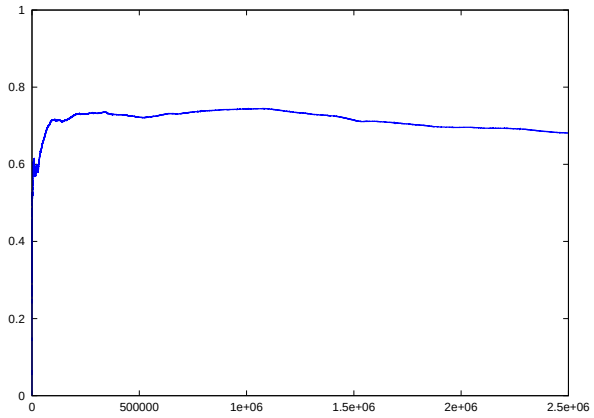
The results of the experiment were very good. We could maintain the average download speed of 4 entry per second, and while maintaining it, our *cumulative harvest rate* (Fig. 4.4) never fell below 0.7. We achieve these results without supervised learning to determine crawling policy (we use supervised learning only to decide whether a document is relevant or not). A number of focused crawling systems use supervised learning, and because of that, when they get to unknown territory, they 'are lost', that is, they have no information about how to proceed, and the *harvest rate* decreases dramatically. This is one of the reasons why large-scale experiments are scarce in the focused crawling literature.

We performed a large-scale experiment in the whole Blogspace, and we downloaded more than 1.6 million web pages in the course of 4 days. After that we performed an even larger crawl, where we downloaded 2.5 million web pages. Figures 4.4 and 4.5 illustrate the performance as a function of documents down-

loaded. It is important to note that because of our flow control mechanism, we



(a)



(b)

Figure 4.4: **The cumulative harvest rate as a function of web pages downloaded** It can be seen that the *cumulative harvest rate* (y axis) remains almost constant and is constantly high as the number of pages downloaded (x axis) grows. (a): First run eventually reaches 1.6 million downloads. (b): Second run eventually reaches 2.5 million downloads. Note that the graphs were generated using all the 1.6 million and 2.5 million points, i.e., they are not smoothed in any way.

are able to maintain an almost constant *cumulative harvest rate*.

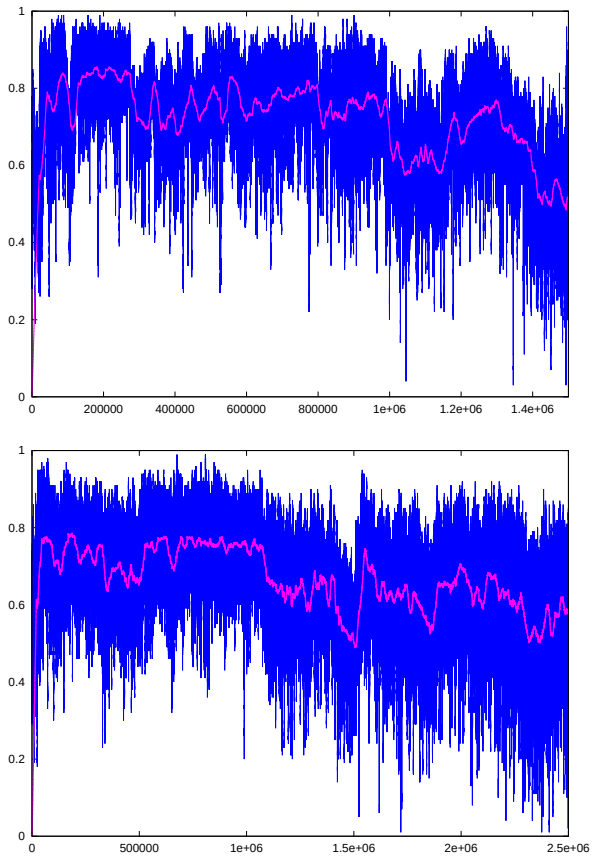


Figure 4.5: **The harvest rate as a function of web pages downloaded** The *harvest rate* (y axis) is constantly high as the number of pages downloaded (x axis) grows. (a): First run eventually reaches 1.6 million downloads. (b): Second run eventually reaches 2.5 million downloads. Note that the harvest rate may drop momentarily, but it then it grows again. (See, e.g., subfigure (a) between $1 \cdot 10^6 \leq x \leq 1.2 \cdot 10^6$.) This is the effect of the flow control mechanism.

4.3 Crawl to follow spreading of information

In order to measure the spreading of information in Blogspace, we have to monitor a number of blogs. Our aim is to cover as many different stories (see below for a definition) as possible, while monitoring as few blogs as we can. One way to do that is to select the most influential blogs in a topic, and monitor them. To do so, one has to identify these blogs. We applied a two step procedure.

First, we crawl a large portion of the Blogspace with our topical crawler, focusing solely on the topic of interest. During this crawl we maintain all the URLs of the blogs that were relevant to the topic in memory, and save all the URLs we encountered and the links on the pages to disk.

Then, after the crawl has finished, we generate a graph of all the relevant blogs encountered by using their URLs and the links between them. Now the task is to find the most influential blogs on this graph. We do this capitalizing on the submodularity of a particular reward function on this graph, as described in the next section.

After we have the most influential blogs in a topic, our task is simple: all we have to do is monitor the most recent posts in these blogs.

4.3.1 The concept of submodularity

In recent years there has been a rapid development in the field of combinatorial optimization, particularly in the maximization of submodular functions. This fast development is due to the speed of the method and that the method has theoretical warranties about the minimal performance it can reach. This seminal method has a 30 year history. It was developed by mathematicians [18] and entered the machine learning community only a few years ago through the work of Kempe et al. [34] on maximizing the influence in social networks. Below, we review the basic concepts of submodularity. A more elaborate description is provided in Sec. 4.3.4. We use the submodularity principle in our searches in Blogspace. In particular, we select the blogs to be monitored by means of this concept. The method can be of relevance in word sense disambiguation, but we have not tried it yet.

The key observation is that under certain conditions (see Sec. 4.3.4), greedy optimization on matroids can provide solutions at least up to a constant factor (e.g., $(1 - 1/e)$, or 0.63) of the optimal solution.

See Sec. 4.3.4 for a short mathematical review of submodularity.

4.3.2 Identifying the most influential blogs in a topic

First, we formalize our problem. We have a graph $G = (V, E)$ of all the relevant blogs encountered, the URLs of their pages and the links between them. The nodes in this graph are the URLs, and the edges are the links. Information from one blog to another propagates if the latter linked to the former, that is, in the reverse direction of the links. We create an information propagation graph $G' = (V, E')$, where the nodes are the same as in G, but the links are reversed.

The notion of 'story' can be formalized as an *information cascade* [4]. The story has a starting URL, and it propagates through the edges of the graph G'. This propagation subgraph is an *information cascade*. The nodes in the information cascade are the different instances of the same story. We would like to detect as much *information cascades* on the graph G' as possible.

The problem is as follows. We can select a set of nodes A on the graph G'. The information cascades that contain those nodes will be detected. We can

only select whole blogs, so we can only increase the set A by all the URLs that are contained on a blog.

We have a reward function $R(A)$, that gives us the goodness of placement A, that is, the number of information cascades detected.[2] We want to maximize this function on the node set V of the graph G', given the constraint that we can only choose n number of blogs to monitor. So, we assign a cost $c(b) = 1$ to all blogs, and we have a budget B. We want to maximize $R(A)$ subject to $c(A) \leq B$, where $c(A)$ is the number of blogs whose URLs are contained in A

It can be shown that the function $R(A)$ is submodular [41], that is, for all placements, $A \subseteq B \subseteq V$ and sensors $s \in V \setminus B$, it holds that

$$R(A \cup \{s\}) - R(A) \geq R(B \cup \{s\}) - R(B). \tag{4.6}$$

Maximizing submodular functions is NP-hard in general [35]. However, it can be shown that for the unit cost case, the following greedy algorithm is near-optimal [18], that is, it achieves at least 63% of the optimum.

We start with the empty placement A_0, and iteratively, in each step k we add the node s_k that gives the maximal marginal gain.

$$s_k = \arg \max_{s \in V \setminus A_{k-1}} R(A_{k-1} \cup \{s\}) - R(A_{k-1}) \tag{4.7}$$

The algorithm stops once it has selected B elements. Note that in our problem, there is no reason to add single nodes one by one. Instead, whole blogs (i.e., all the nodes of a blog) are to be added in each step. This change does not affect the algorithm, which otherwise remains the same.

4.3.3 Experiments: determining the most influential blogs in a topic

We downloaded more than 1.6 million web pages on more than 4000 blogs using our topical blog crawler. After that, we created the information propagation graph G' described in the previous section.

In this graph we considered all the non-trivial information cascades (i.e., those that contain at least two nodes), but when finding those cascades we did not follow through hubs in the information propagation graph. In this aspect, our procedure is different from that of [41]. The reason for this is that hubs (in our case, pages with more than 200 outgoing edges) are not likely to be blog entry pages. Aside from everyday experience, one can suspect this experimentally: when we followed the hubs, many implausible information cascades are created, that have $10^6 - 10^7$ nodes. These huge cascades are due to entering even larger hubs step-by-step and loosing focus. This conjecture could be easily proven by analyzing these huge cascades in details. We did not follow the hubs and these huge cascades did not appear.

[2] We note that $R(A) = F(A)$ of Eq. 4.17 if $\pi(0)$ equals to the number of all cascades. However, this number is not known. In [41] a sufficiently large number was chosen, so $R(A) + C = F(A)$, where C is unknown. This constant C should have no effect on the algorithm.

We defined a budget $B = 100$ for the algorithm. Note that the algorithm gives a near-optimal solution at every intermediate step, i.e., for smaller step numbers, too. Figure 4.6 shows the proportion of cascades found at every step. It can be seen that monitoring 20 to 40 blogs is enough to detect 80 to 90 percent of the cascades.

In Table 4.1, we show the first 20 blogs the greedy algorithm found, the number of cascades detected by them, and their relevance rating. The number of cascades detected are calculated as the algorithm progresses. For example, the number of cascades detected for the second blog means the number of *additional* cascades detected after selecting the first blog. Note that the most influential blogs are not the same that have the highest relevance rating. For comparison, in Table 4.2, we have included the 20 most relevant blogs, and their relevance rating.

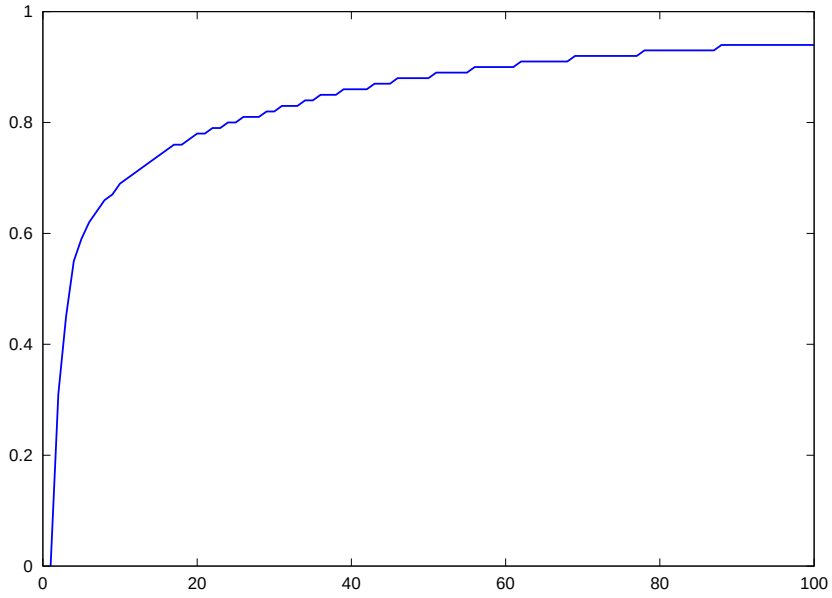


Figure 4.6: **The proportion of cascades detected as a function of the number of blogs** It can be seen that as the number of blogs we monitor (x axis) increases, the proportion of cascades detected (y axis) increases very quickly at first, then it levels off. This is a consequence of submodularity. It can also be seen that monitoring 20 to 40 blogs is enough to detect 80 to 90 percent of the cascades.

blog name	cascades covered	relevance
daily.greencine.com	3093	0.77
www.impawards.com	1380	0.89
slate.msn.com	991	0.46
oggsmoggs.blogspot.com	415	0.80
opalfilmsarchive.blogspot.com	314	0.56
carson83.blogspot.com	162	0.69
bleeding-tree.blogspot.com	159	0.41
templeofschlock.blogspot.com	158	0.46
www.geocities.com	148	0.42
metaphilm.com	125	0.52
kamikazecamel.blogspot.com	114	0.48
diyfilmmaker.blogspot.com	102	0.40
actionflickchick.com	93	0.84
mysterymanonfilm.blogspot.com	87	0.83
blogs.amctv.com	84	0.57
ferdyonfilms.com	76	0.92
www.chutry.wordherders.net	70	0.65
he-shot-cyrus.blogspot.com	66	0.48
www.premiumhollywood.com	61	0.56
www.horror-101.com	61	0.67
Cumulated number of cascades detected	7759	

Table 4.1: **The 20 most influential blogs in the 'movies' topic**
We show the first 20 blogs the greedy algorithm found, the number of cascades detected by them, and their relevance rating. The number of cascades detected are calculated as the algorithm progresses. For example, the number of cascades detected for the second blogs means the number of *additional* cascades detected after selecting the first blog. Note that the most influential blogs are not the same that have the highest relevance rating (Table 4.2).

blog name	relevance	
stickyfloor.blogdrive.com	1.0	
moviesblog.mtv.com	1.0	
moviepalace.blogspot.com	0.99	
adnauseum.blogdrive.com	0.99	
www.combustiblecelluloid.com	0.98	
vergingwriter.blogspot.com	0.98	
distantorigin.blogdrive.com	0.97	
jadedviewer.com	0.97	
www.greencine.com	0.95	
www.reelviews.net	0.95	
filmfreakcentral.blogspot.com	0.95	
amusicment.blogspot.com	0.95	
criticalcorner.net	0.95	
www.dvddrive-in.com	0.94	
www.thefilmpanelnotetaker.com	0.94	
www.breabennett.name	0.94	
ferdyonfilms.com	0.92	
www.arabfilm.com	0.92	
www.horrorwatch.com	0.92	
mrpeelsardineliqueur.blogspot.com	0.91	
Cumulated number of cascades detected		523

Table 4.2: **The 20 most relevant blogs in the 'movies' topic.**
Note (1) the entries are very different from the 20 most influential blogs and (2) the cumulated number of cascades that can be detected from these blogs is **only** 523. (Table 4.1)

4.3.4 A short mathematical review of submodularity

Below, we provide the mathematical definition of submodularity and provide examples for a few applications.

Notation: Let $V = \{1, \ldots, n\}$ denote a finite set of n elements. Without loss of generality, we assume that we have function F over the set with the following properties: $F(\emptyset) = 0$ and we say that F is *normalized*.

matroid: Given a V finite basic set. Set of subsets of $\mathcal{F}$ [more precisely the pair $(V, \mathcal{F})$, where $\mathcal{F} \subseteq \mathcal{P}(V)$] is called *matroid* if it exhibits the following three properties:

1. the set is not empty;
$$\mathcal{F} \neq \emptyset \tag{4.8}$$
2. the set is independent, i.e., all subsets of an *independent set* (= elements of $\mathcal{F}$) is also independent;
$$\forall X, Y \in \mathcal{P}(V) : [(X \subseteq Y \wedge Y \in \mathcal{F}) \Rightarrow X \in \mathcal{F}] \tag{4.9}$$
3. independent subsets can be expanded mutually: we can choose an element from a larger set that is not element of the smaller set and can add it to the smaller set and the expanded smaller set remains independent.
$$\forall K, N \in \mathcal{F} : [|K| < |N| \Rightarrow \exists x \in N - K : (K \cup \{x\} \in \mathcal{F})] \tag{4.10}$$

Example:

1. $\mathcal{F}$:={linearly independent columns of matrix A}; note: this is the origin of the word 'independent'.

One can show that under certain conditions, greedy optimization on matroids can provide solutions at least up to a constant factor (e.g., $(1-1/e)$, or $1/2$) of the optimal solution.

monotone/monotonic function: Set function $F : 2^V \to \mathbb{R}$ is *monotone*, if for any $A \subseteq B \subseteq V$
$$F(B) \geq F(A). \tag{4.11}$$
For example, entropy for discrete variables: $F(A) = H(x_A)$, information gain [(4.16)], set cover [(4.20)], rank of matroid.

symmetric function: Set function $F : 2^V \to \mathbb{R}$ is *symmetric*, if for any set $A \subseteq V$ $F(A) = F(V \setminus A)$. Example: mutual information.

positive/non-negative function: Set function $F : 2^V \to \mathbb{R}$ is positive if for any set $A \subseteq V$ $F(A) \geq 0$.

submodular function

- Definition:
  - Def1: Set function $F : 2^V \to \mathbb{R}$ is called *submodular*, if for any pair of sets $A, B \subseteq V$

$$F(A \cup B) + F(A \cap B) \leq F(A) + F(B). \tag{4.12}$$

  - Def2 ('adding a new observation, the smaller subset gains more more information'; $\Leftrightarrow$Def1): for any set pairs $A \subseteq B \subseteq V$ and for any $s \in V \setminus B$

$$F(A \cup \{s\}) - F(A) \geq F(B \cup \{s\}) - F(B). \tag{4.13}$$

- Properties:
  - submodular functions are closed for linear combinations.
  - If F_1 is submodular on V_1 and F_2 is submodular on V_2, then

$$F : S \subseteq V_1 \cup V_2 \mapsto F_1(S \cap V_1) + F_2(S \cap V_2) \tag{4.14}$$

    submodular $V_1 \cup V_2$-n.
  - Let

$$F(A) = g(|A|), \tag{4.15}$$

    where $g : \mathbb{N} \to \mathbb{R}$. Then: F is submodular $\Leftrightarrow$ g is concave.

Examples:

1. Feature selection (Naive Bayes Model): Let y ('sickness'), $x_1, \ldots, x_n$ ('symptoms') be stochastic variables. W look for the most informative k pieces of features, that is:

$$A^* = \arg \max_{A: A \subseteq V, |A| \leq k} F(A) := IG(x_A; y) := H(y) - H(y|x_A) \tag{4.16}$$

   One can show that if x_is are conditionally independent given y then F is submodular.
2. Blogspace:

$$A^* = \arg \max_{A: A \subseteq V, c(A) \leq B} F(A) := \pi(\emptyset) - \pi(A), \tag{4.17}$$

   where $\pi(\cdot)$ tells my cost if I choose its argument. $\pi(\emptyset)$ is a large fixed number. For example:
   (a) $\pi(A)$=E[the number of unobserved info cascade if I choose A],
   (b) $\pi(A)$=E[the time that passes until I observe the info-cascades if I choose A]
   (c) $\pi(A)$=E[the number of blogs reached by the info cascade before reaching A]
   (d) linear combinations of these.

Here:

$$c(A) = \sum_{s \in A} c(s), \tag{4.18}$$

where $c(s)$ is the non-negative cost of the states (= the blogs). *Unit cost*: when all blogs have the same cost, ($c(s) = 1$). *Non-constant cost*: costs may differ.
There is an efficient algorithm for normalized, monotone, submodular function that warrants $\frac{1}{2}(1 - 1/e)$ portion of the optimum. It is called cost effective lazy forward selection (CELF) algorithm [41].

3. Factorizing distributions (structure learning): Let $x_1, \ldots, x_n$ be stochastic variables. The task is to partition the variables into 2 groups with minimal mutual information:

$$A^* = \arg \min_{A: A \subseteq V, 1 < |A| < n} F(A) := I(x_A, x_{V \setminus A}), \tag{4.19}$$

where $I(x_A, x_B) = H(x_B) - H(x_B|x_A)$.

4. Set cover function: $A \subseteq V$, $F(A) :=$ the volume covered by sensors at places of A, that is:

$$A^* = \arg \max_{A: A \subseteq V, |A| \leq k} F(A) := |\cup_{i \in A} S_i|. \tag{4.20}$$

5. Cut function: cost of cut in a graph, that is

$$A^* = \arg \min_{A: A \subseteq V, 1 < |A| < n} F(A) := \sum_{s \in A, t \in V \setminus A} w_{st} \tag{4.21}$$

4.4 Detecting the most influential blogs based on document content

In the previous semester, we have planned an experiment to detect influential blogs based on content. In the preceding experiments based on the work of Leskovec et al. [41], it was taken granted that if a blog post links to another post, then information has spread between the two posts. We would like to test this assumption by measuring the similarity of posts with a link between them, and then thresholding the graph of information spreading (i.e., removing all the edges where the similarity between two posts is less than the threshold) before running the algorithm.

This way we will obtain information cascades where information has certainly spread between the blog posts in the cascade. If we do not perform this thresholding, it may be the case that the two blogs are linked only because, for example, one has a link on its menu bar to the other, but they do not share content.

In the last semester, we could not complete the experiments due to memory problems. In this semester, we have successfully completed it using our new architecture based on Lucene.

We have used 1.5 million webpages collected with our crawler, the same amount we used in the 4th semester. We have taken the graph where the nodes are the webpages, and the edges are the links between them. We have compared all pairs of webpages with a link between them by the cosine distance of their Bag of Words representation. A histogram of the distribution of the result can be seen in Fig. 4.7. It can be seen that most of the links have a low similarity value; the distribution is skewed to the right. Based on this data, we can not say that a link from one document to the other implies that they share significant content.

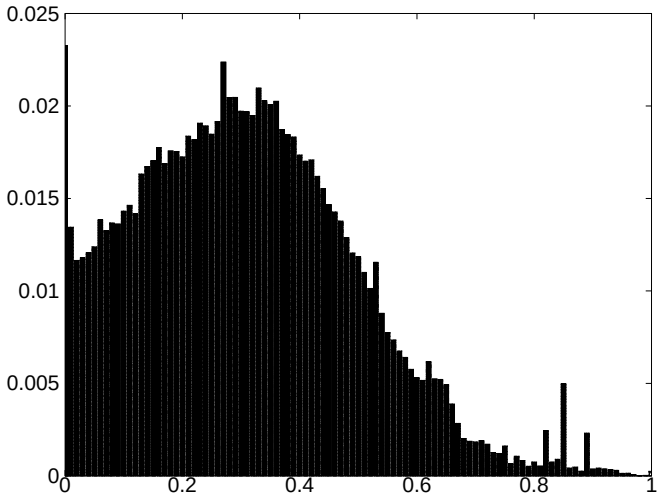


Figure 4.7: **A histogram of the distribution of similarity values on links** We have taken the graph of 1.5 million webpages where the nodes are the pages, and the edges are the links between them. We have compared all pairs of webpages with a link between them by the cosine distance of their Bag of Words representation. The similarity values are on the horizontal, their relative frequency on the vertical axis. The first bin contains all the values v with $0 <= v < 0.01$, etc. It can be seen that most of the links have a low similarity value; the distribution is skewed to the right. Based on this data, we can not say that a link from one document to the other implies that they share significant content.

We have repeated the experiments we conducted in the 4th semester (See Sec. 4.3) with and without thresholding. The experiments without thresholding are conducted in exactly the same way as those in the 4th semester. The

results were also very similar (Fig. 4.8). In the other experiment we deleted all the links where the similarity between the endpoints was less than 0.5. It is somewhat surprising that the results did not change significantly (Fig. 4.9), except for the fact that there were less cascades (2748 vs 7471 in the case without thresholding).

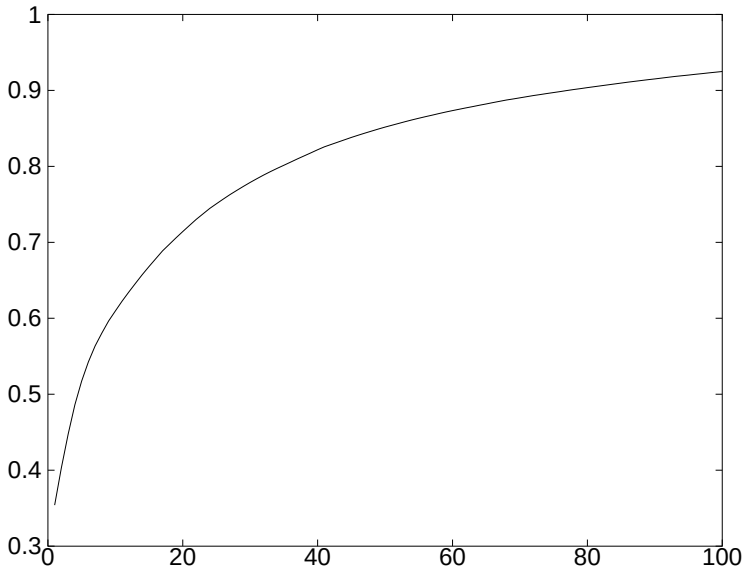


Figure 4.8: **The proportion of cascades detected as a function of the number of blogs - not thresholded** The results are consistent with our experiments in the 4th semester, although the graph is a little less steep at first. There were 7471 cascades. It can be seen that as the number of blogs we monitor (x axis) increases, the proportion of cascades detected (y axis) increases quickly at first, then it levels off.

However, if we look at the blogs that were deemed most influential (Tables 4.3 and 4.4), a significant change can be spotted immediately. The blogs in the two tables are different. One of the most striking examples of the dangers of disregarding content is the blog `www.espejopintado.com`. It is present in the non-thresholded table, but it is a blog in Spanish! It is not present in the version where we have thresholded the links, because it was eliminated based on its content. It is probably the case that many other blogs link to it on their sidebars or as menu items, despite the fact that it is Spanish. Comparing content can help reduce the effect of such artifacts.

blog name	cascades covered
screenrant.com	2645
gatewaycinephiles.com	367
www.girishshambu.com	332
www.dvdjournal.com	289
thaifilmjournal.blogspot.com	231
moviescreenshots.blogspot.com	189
www.jonathanrosenbaum.com	157
www.combustiblecelluloid.com	128
mysterymanonfilm.blogspot.com	118
livingincinema.com	98
chrisbourne.blogspot.com	97
www.impawards.com	91
www.makingof.com	86
www.simpleweblog.com	85
movieprojector.blogspot.com	79
criticafterdark.blogspot.com	77
bwanavoodoo.wordpress.com	77
www.horror-101.com	64
www.reverendphantom.com	64
www.espejopintado.com	62
Cumulated number of cascades detected	5336

Table 4.3: **The 20 most influential blogs in the 'movies' topic - not thresholded**
We show the first 20 blogs the greedy algorithm found, the number of cascades detected by them, and their relevance rating. The number of cascades detected are calculated as the algorithm progresses. For example, the number of cascades detected for the second blogs means the number of *additional* cascades detected after selecting the first blog. One striking example of the need of filtering the links based on the content of documents is the blog `www.espejopintado.com`. It is present in the table, but it is a blog in Spanish!

blog name	cascades covered
jadedviewer.com	446
www.girishshambu.com	318
www.alltooflat.com	157
screenrant.com	152
www.makingof.com	73
thaifilmjournal.blogspot.com	73
blog.nicksflickpicks.com	69
www.espejopintado.com	65
www.reverendphantom.com	59
www.mattriviera.net	56
colemancornerincinema.blogspot.com	52
mikesyoutalkingtome.blogspot.com	50
mysterymanonfilm.blogspot.com	42
movieprojector.blogspot.com	40
criticafterdark.blogspot.com	33
www.1000misspenthours.com	31
the-black-glove.blogspot.com	31
www.if.com.au	29
thevaultofhorror.blogspot.com	28
fourofthem.blogspot.com	28
Cumulated number of cascades detected	1832

Table 4.4: **The 20 most influential blogs in the 'movies' topic - thresholded**
We show the first 20 blogs the greedy algorithm found, the number of cascades detected by them, and their relevance rating. The number of cascades detected are calculated as the algorithm progresses. For example, the number of cascades detected for the second blogs means the number of *additional* cascades detected after selecting the first blog. The Spanish blog is not present here.

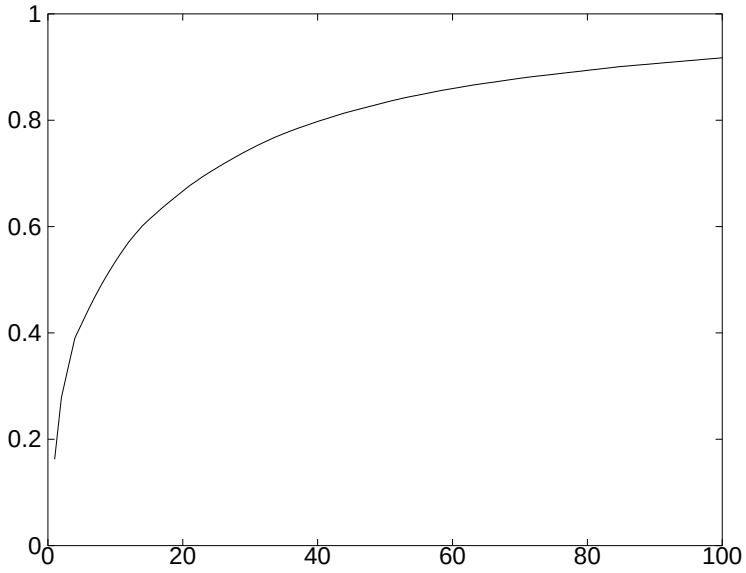


Figure 4.9: **The proportion of cascades detected as a function of the number of blogs - thresholded** The results did not change after thresholding, despite the fact that there are much less cascades left (2748 vs 7471 in the case without thresholding). It can be seen that as the number of blogs we monitor (x axis) increases, the proportion of cascades detected (y axis) increases very quickly at first, then it levels off.

Chapter 5

Progress in interpreting text

5.1 Extracting word graphs from text

As advocated in the introduction, our approach towards representing semantic content of documents is through keyword graphs, that can be thought of as simple ontologies. The use of keyword graphs to decide semantic similarity of documents is twofold: (i) we wish to represent background knowledge collected from a corpus of documents as a large word graph, and (ii) we wish to represent single documents as small keyword graphs. We aim to decide document similarity by comparing small keyword graphs extracted from documents also utilizing the large background knowledge graph during the comparison process.

The core of the task is to extract word graphs, i.e. word-to-word relations from text. Our ontology will contain (possibly weighted) words as nodes, and labelled and weighted edges between words. The edge labels designate relations between words coming from a variety of sources, as will be explained below. Our main objective is to represent blog entries, which are relative short documents, thus we need to extract every bit of meaningful information from them.

Our original methods for keyword graph extraction developed in a previous USAF project are not satisfactory for this purpose and need to be augmented by several novel components. The core of the original method built on statistical techniques, finding *often co-occurring* words in texts. The use of such statistical methods is only viable on a *large* body of texts, since they operate by averaging out noise. For example our methods for finding word-to-word relations by examining the relative frequencies of words co-occurring in a specified window size were satisfactory for long scientific documents. However, in the case of blogs, we must prepare for smaller and less well organized pieces of text, thus more exact methods are required, that introduce as little noise (false word-to-word relations) as possible.

In order to do so, we revised our ideas about keyword graph extraction, and came up with a more decent document processing system. As already mentioned, the aim of the system is to extract as many true word-to-word

relations as possible. We categorized the relations of interest in the following manner:

1. **Morphological relations**: derived and inflected forms of the same word in different text positions should be recovered. This enables joining their contexts, which helps overcome the problem of data sparsity (one word form might occur only a few times in a document, but its derived forms might occur as well; for example *run, runs, running, ran*).

2. **Syntactic relations**: syntactic dependencies are of primary interest, since they reflect semantic relations between words that define the topic of the document. The recovery of exact syntactic relations helps overcome the noise problems introduced by statistical methods that look for word co-occurrences in a given text window and can be compromised if textual data is not sufficient. We employ dependency parsing to recover syntactic relations:

   (a) **Dependency relations**: Dependency parsing of sentences results in *labelled* relations between words. Each word may have a syntactic head it depends on, i.e. modifies or influences its meaning. For example in the sentence *The monkey ate the banana.*, the *monkey* is the *subject* of *ate*, and the *banana* is the *object*.

3. **Semantic relations**: Semantic relations between different entities also help joining their contexts and thus decrease the problem of data sparsity.

   (a) **Background knowledge from semantic corpus**: When we encounter a word in a text, we retrieve a lot of extra information based on our semantic knowledge about the concept that it represents. For example, we know that a *river is a kind of water*, thus when we talk about rivers, we also reason about it using the properties of water. This kind of extra information might be utilized in automatic document content comparison, by extending keyword graphs with word relations from semantic corpora, such as WordNet.

   (b) **Pronominal anaphora resolution**: We often use structures in which we refer to previously mentioned entities without explicitly naming them again (e.g. by using pronouns). These references might as well hide useful word relations. For example in the sentences *The monkey ate the banana. It was very delicious.* the word *it* refers to the *banana*, and *delicious* is the antecedent of *it*, thus, *delicious* should be related to the *banana*.

   (c) **Non-pronominal coreference resolution**: Coreference resolution is a task similar to anaphora resolution. Two words corefer if they refer to the same entity. So every anaphora is also a coreference; for our purposes the two are equivalent. One special form of coreference is when none of the two words is a pronoun. For example *John F.*

Kennedy and *the President of the United States* might refer to the same person.

Let us have a look at the following excerpt from an imaginary text. How many word-to-word relations can we extract from it?

> *There was a small watermill in the valley. It was quite hidden between the mountains. It is rare to see such buildings on the Danube nowadays. The river was flowing very fast at this region of the country. The miller has just arrived...*

Ideally, we would like to end up with a word graph roughly like this:

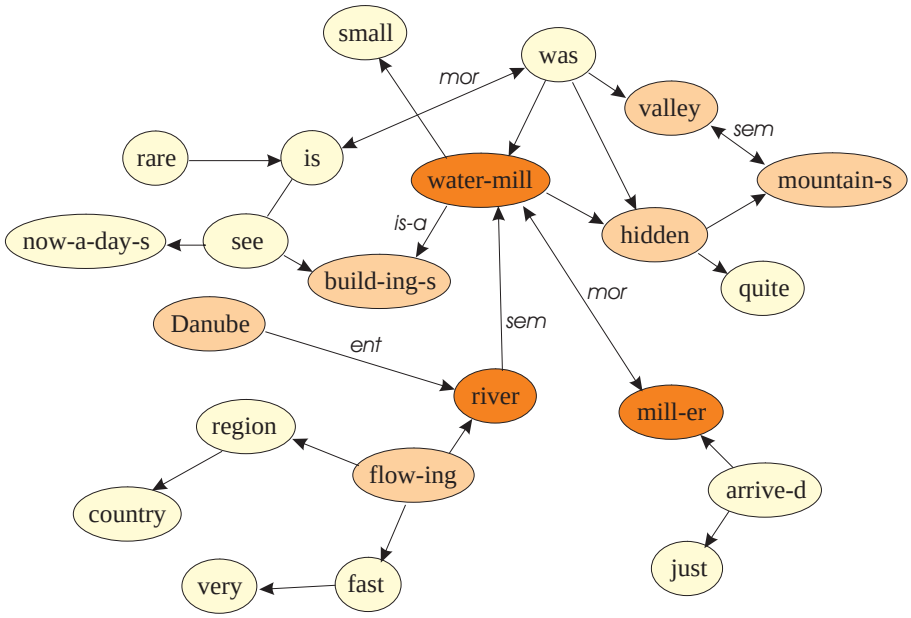


Figure 5.1: **Example keyword graph of the quote.** Node colors encode word weights; darker color means more important word. Arrows represent the direction of word relations. Edge labels denote relation types (*mor*: morphological relation, *sem, is-a*: semantic relation *ent*: named entity. Syntactic labels are omitted here for legibility).

The determination of word weights will be the result of a keyword extraction procedure operating on the extracted word graph.

In what follows, we provide a detailed description of the word-relation recognition techniques enumerated above. Some of the methods were developed by ourselves, others were obtained from publicly available sources in order to integrate them to our system. We have examined state-of-the art language pro-

cessing packages in order to find recent text processing techniques and obtain readily available software tools.

5.1.1 Morphological analysis

The task of morphological analysis is to break up a word into its constituent morphemes (basic entities that build up words in a language). The morphemes may designate word stems of derivational and inflectional affixes (prefixes and suffixes in English).

Our use of morphology in word graph extraction is twofold

- **Lemmatization**: inflectional suffixes do not change the meaning of words, they are used to express case, tense, plurality, etc. instead. Obtaining the lemma of a word enables us to recover different word forms of the same lexeme, i.e. different word forms with roughly the same meaning. For example *run* and *runs* (as a verb) mean the same, the difference is that the latter is in third person singular. Inflection might also be expressed by irregular forms, like *bring* - *brought* or *child* - *children*. However, obtaining the lemma is not only about stripping some suffixes and checking a list of irregular words. Consider the words *underwent* or *firemen* where the lemmas *undergo* and *fireman* are more complicated to obtain: we must consider prefixing and compounding). Also the correct lemma of a word may depend on its part of speech in the actual sentence. For example *colored* as a verb has the lemma *color* (*ed* is an inflectional suffix denoting past tense), but as an adjective, its lemma is *colored* (*ed* here is a derivational suffix).

- **Derivational relatedness**: by derivation we create new words from existing ones. The new word usually means something else, but it is often strongly related to the one it is derived from. For example *simplify* means to make something *simple*, and *simplification* is the process itself.

  We may create new word forms (in English) by suffixing (*simple-ify*), prefixing (*over-simplify*) and compounding (*fire-man*). In the case of compounds, the compounded word is also (semantically) related to its parts: *fireman* is a kind of *man* and is related to *fire*. Morphological analysis should be able to recover these relations between words in order to infer semantic relatedness between them and their contexts. For example, we should be able to realize that the two sentences *'He oversimplified the problem.'* and *'The problem became too simple.'* essentially convey the same meaning.

Morphological analysis for languages other than English might be of even more importance. For example highly inflecting languages (like Hungarian and Finnish) have many more forms of the same word. Other languages (like German) make use of excessive compounding to create very long words by joining

parts that are themselves affixed. Some (e.g. Arabic) languages express complex syntactic structures by morphology, resulting in words that could be only expressed by longer phrases in English.

One popular approach to morphological analysis in Natural Language Processing is the use of stemming algorithms (e.g. Lovins, or Porter stemmer). These algorithms are simple and fast, but have a number of inadequacies since they operate by stripping away word endings. They:

- do not handle prefixes (e.g. *un-related*)
- do not handle compounds (e.g. *dog-house*)
- distort word endings (e.g. *generate* becomes *gener*)
- do not take the Part of Speech of the word into account (e.g. *colored* as a verb and as an adjective)
- do not output multiple possible analyses of a word (e.g. *building* can be a noun or a continuous verb)
- do not detect case/tense markers (e.g. *treated* is the past tense of *treat*, or *his* is third person singular and possessive).

We have examined the literature for existing approaches to morphological processing, and looked for off-the-self tools that could be utilized here. There are many readily (and freely) available software packages that strip away word endings, lemmatize words or do a little deeper analysis. Commercial products also exist that perform more detailed analysis. Also, there exist techniques for *learning* morphological analysis for various languages. However, we did not find any freely available, off-the-self system that satisfied our requirements listed above, so we decided to implement our own morphological analyzer. Its details are deferred to Sec. (5.1.8). Our system provides the following functionalities:

- handles prefixes, suffixes and compounds and their combinations without distorting word endings
- can lemmatize words, handles irregulars even in combination with prefixing and compounding
- determines case and tense markers even for irregulars and closed class words (e.g. pronouns)
- can enumerate multiple possible analyses of a word, also taking the desired Part of Speech into account
- provide hypothetical analysis for unknown words (e.g. *un-trool-ed*)

Moreover, the implementation is relatively fast, it analyzes about 20,000 words per second on current computers, and consumes low memory.

We have *tested our morphological analyzer* on a set of related word pairs extracted from WordNet. WordNet contains word-to-word relations labelled as 'Derivationally Related'. We have generated a list of word pairs by enumerating all such relations (9282 in total). Also, WordNet is capable of lemmatizing words that are contained in it. Utilizing this functionality, we have also generated a list of word-lemma pairs (37262 in total). Our analyzer *recognized 90% of the derivationally related word pairs* (see table 5.1), and *lemmatized more than 99% of the listed words* correctly. Note that relying solely on WordNet for morphological analysis would not be sufficient, since it is unable to handle newly created words that are not contained in WordNet (or any manually created dictionary), *but occur often in blogs.*

Analyzer	No derivation	Different root	Same root	Accuracy
FreeLing	0	8142	1140	12%
morph	0	7025	2257	24%
pc-kimmo	423	1137	7722	83%
Our analyzer	0	880	8402	90%

Table 5.1: **Morphological derivation comparison.** Each analyzer was given pairs of words with common roots from WordNet. The accuracy of an analyzer is measured by how often it is able to find a derivation for both words in a pair, and how often the derivations have a common root.

5.1.2 Dependency parsing

Syntactic structures in sentences often reflect semantic relations between concepts, thus uncovering syntactic relations between words of a sentence are of particular interest when characterizing the content of documents. There are many forms of syntactic analysis, from *deep parsing* like *phrase structure parsing*, that results in a complete structural analysis of a sentence, to *shallow parsing* that looks for typical simpler structures like noun phrases or structured entities like dates. Somewhere between the two is *dependency parsing*, which aims to recover dependency relations between words of a sentence. For example a verb may have its subject and its object as its dependents. A noun may have an adjective and a determiner as its dependents. These dependency relations often reflect semantic relations between words. For example, in the phrase *'the blue car'*, the adjective *blue* is a dependent of the noun *car*. Semantically, *blue* is related to *car* by denoting its color.

In dependency relations, the relation is always between a *head* word and a *dependent* word. In most cases this dependency means that the dependent word modifies or clarifies the meaning of the head word, provides more detail about it. Thus, we intuitively found this kind of relation adequate for building semantic graphs describing the meaning of documents.

One popular readily available method for dependency parsing in the literature is the method of Nivre et. al. [49] Their parser, called the Malt parser

has been successfully applied to many languages including English, and has a publicly available Java implementation that we integrated into our system. Howewer, it does not take raw text as its input: it operates on tokenized, part-of-speech tagged sentences, not on pure text. Fortunately, another freely available tool, the Stanford Part-of-Speech tagger does the preprocessing job we need: it tokenizes sentences into words and punctuation signs, and attaches part-of-speech tags to them as well. Both systems had been trained on English texts, utilizing the same part-of-speech tag set, called the Penn tag set. This enabled us to link the two to form one system that processes pure text and delivers dependency relations.

5.1.3 Anaphora resolution

Anaphors refer to entities in other sentences or different parts of the same sentence. Anaphora resolution highly relies on the semantics of the words. In the example '*The monkey ate the banana. It was very delicious.*', the word *it* may refer to the *monkey* and to the *banana* as well. The only cue to decide which it truly refers to is the meaning of the words *monkey* and *banana*. Usually a *banana* can be delicious, and, while a *monkey* could also be, it is clear that the monkey eats the banana and not the opposite, thus *it* should refer to the *banana*. In another example '*The boy and the girl were sitting next to each other. She didn't say a word.*', the pronoun *she* refers to the girl, and not to the boy, because of gender agreement. Deciding gender for pronouns is easy, while doing so for general nouns like *boy*, *father*, *guard*, etc. is not, and again, requires some semantic knowledge.

Anaphora resolution systems build on grammatical analysis of a sentence to enumerate possible referents. They often require detailed analysis of sentences, like complete or partial phrase structures. Some readily available systems operate on grammatically analyzed input, or employ a parsing module. The systems we decided to use, OpenNlp and JavaRap, both rely on phrase structure parsers. The former relies on the OpenNlp parser, the latter on the Charniak parser.

See also the section on incorporated software in Sec. 5.1.9)

5.1.4 Non-pronominal coreference resolution

The task of non-pronominal coreference resolution is similar to pronominal anaphora resolution: names or words referring to the same entity must be recovered. However, the solution methods may be quite different, since recognition of coreference between named entities requires even more knowledge about the meaning of the entities.

5.1.5 Acronym resolution

Acronyms, like USA or FBI might also be resolved by utilizing a database of such abbreviations, but care must also be taken to handle ambiguous acronyms. In that case, the context of the acronym must also be taken into account for

proper resolution. Furthermore, new acronyms are created on a daily basis that a database can not contain, so algorithms for on the fly resolution of acronyms may also be needed.

5.1.6 Keyword extraction

To further compress the graph representation of documents, we may reduce the size of the graphs by keeping only the most important words and relations between them. The graph representation seems promising for finding important words. Building on this representation, we have devised a new method for extracting keywords.

In our new approach to keyword extraction, a word is important if it is *supported* by its context. Consider a document that is about *artificial neural networks* for example. If the document mentions a lot of terms related to artificial neural networks then the words *neural* and *network* will be considered important words in the document, even if they do not occur very often. As an opposing example, if a word occurs quite frequently in a document, but is not related to the main topic of the document (for example, 'let us *denote*...' occurs often in mathematical texts, yet *denote* is rarely a keyword), it should not be treated as an important word, since it does not receive support from its context. Thus, we can conclude that it is exactly the graph structure of word-to-word relations that can be used to decide the importance of the word.

Relaxation algorithms

Diffusion and relaxation algorithms can be used to propagate node activations on weighted graphs. These algorithms are essentially the mathematical implementations of the idea of the *contextual support* described above. Suppose that we are given a word graph with weighted edges and nodes extracted from a document by inserting edges as described in the previous sections. Edge weights represent connectivity (i.e. relation) strengths between words. Node weights represent word importance. Informally, our algorithm for keyword extraction performs the following: starting from an initial set of word weights set by the frequencies of the words in the document, it relaxes the weights by spreading activation through the edges between words.

More formally, let us denote the vector of word weights by $\mathbf{w}$, and the matrix of edge weights by G, the algorithm performs the following iteration (t denotes time index):

$$\mathbf{w}_{t+1} = \sigma\big((1-\alpha)G\mathbf{w}_t + \alpha\mathbf{w}_0\big)\ , \tag{5.1}$$

where α is some smoothing value in $[0,1]$, and σ is some nonlinear function, for example it may represent renormalization of the vector to prevent the iteration from diverging. Activation spreading is implemented by multiplying weights $\mathbf{w}$ in time step t by the edge-weight matrix. Note, that if $\alpha = 0$, i.e. the iteration is not driven by the original weights but it is only used in the initialization. Furthermore, if the matrix G is stochastic and ergodic and σ is the identity function, than the matrix G and the vector $\mathbf{w}_0$ define a Markov chain, and the

iteration converges to the (unique) stationary distribution of that Markov chain. The algorithm then resembles to the PageRank algorithm used to calculate the importance of documents on the Internet.

5.1.7 Feature representation of words based on relations

An ontology extracted from a large body of texts enables the characterization of a word's meaning through its relations. A popular representation of entities, called feature representation can be generated from such an ontology. This representation embeds words to the space of word-to-word relations, and results in real valued vector representation for each word. Feature representations are useful for a number of machine learning algorithms. For example, a simple application could be to decide word similarity or to cluster words based on similarity. Similarity in feature space might be computed for example by taking the cosine of the angle of two vectors (dot product).

Here, we describe one way to generate feature representations from word graphs. We map each word to a real valued vector according to the following. Let N be the number of words, and K be the number of relation types in the word graph. The feature vector of a word w is a vector of $2K$ parts, each part containing N entries:

$$F_w = [v_1; \ldots; v_K; v^1; \ldots; v^K] \ , \tag{5.2}$$

where the first K parts represent the weights of the outgoing edges from word w, and the second K parts represent the incoming edges of word w, each kind of relation taking one part. For example the N values in v_k are the weights of outgoing edges being labelled by the kth relation type for each of the N possible target words in the graph. Of course, when no edge exists between two words with the specified label in the graph, the corresponding value is 0. Thus, this representation will be quite sparse, since most words are only connected to a fraction of the other words in the graph, and only with some kind of the possible relation types.

5.1.8 Details of our morphological analysis algorithm

Overview

In English, the task of a morphological analyzer is to find prefixes and suffixes attached to one or more stems. Suffixing is used most extensively to alter the meaning of the word and derive new ones. In most cases, they change the part-of-speech of the word to enable usage in a different grammatical role. Of course, the meaning of the word changes in this case, however, the two words are often strongly related, for example *simple* and *simplify*.

There can be many ambiguities in morphological processing, at many levels. For example, a word might be segmented in more ways into morphemes. Also, the morphemes may have more than one morphological roles, e.g. *ant* can be a stem and a suffix too. Third, a segmentation can be interpreted in more than

one way, e.g. the suffix *ed* may be the indicator of past tense and also may be used to derive an adjective.

Thus, our morphological analyzer aims to perform all of the following tasks:

- handle prefixes, suffixes and compounds, recognize known stems
- supply all possible segmentations of a word
- supply all possible interpretations of a segmentation
- supply inflectional information (tense, case) even for regular words
- supply a hypothetical interpretation for words with unknown stems and known affixes

The algorithm works by first segmenting the word into possible sequences of morphemes by finding all morphemes contained in a word and trying to chain them after one another and filtering morphemes that do not fit into a chain. Then, it enumerates all possible legal interpretations of a segmentation, filtering them by applying affix attachment rules.

Data used to build an analyzer

The analyzer is built from the following pieces of information.

- **List of words**: a list of English words along with their possible part-of-speech tags. The list also contains derived forms of a word. It has been generated with the use of a corpus of texts and a lexicon. A word was incorporated into the list if it occurred in the lexicon and occurred in the corpus at least three times.
- **List of affixes**: a list of prefix and suffix rules. A rule is of the form

  `affix lemma_PoS-derived_PoS.inflection`,

  where `affix` is a prefix or a suffix, `lemma_PoS` is the part-of-speech of the word to which `affix` can be attached to, and `derived_PoS` is the part-of-speech of the resulting derived word, and `inflection` is the list of inflection flags of the resulting word. One affix may have more then one interpretations, i.e. having more than one `lemma_PoS - derived_PoS` pairs. For example, the rule

  `ing v-n v-v.cont`,

  means that the suffix `ing` may derive a noun from a verb or may create a continuous verb from a verb.
- **List of morphological (rewrite) rules**: these rules can be applied to words to alter their ending to enable the attachment of suffixes. The rules are of the form

  `endin_str rewrite_str`,

where `ending_str` is the word ending to which the rule can be applied, and `rewrite_str` is the string which is attached to the word, after the ending is deleted. The strings may contain the capital letters `C` and `V` to denote any *consonant* or *vowel*. For example the rule

`Cy Ci`,

means that if a word ends with a letter `y` after a consonant, than the `y` may be rewritten to an `i`, like in *imply* → *implication*.

A rule also may express for example that the last consonant must be doubled if the previous letter was a vowel

`VC VCC`,

like in *big* → *bigger*.

- **List of regular words**: lists containing regular past tenses for verbs, plurals for nouns, and inflection for adjectives.

Algorithm details

In the following, we describe the steps of the algorithm in detail.

- **Recognition of possible morphemes:** A string retrieval tree data structure (trie) is built from the (words) stems and affixes contained the above described lists. Upon reading a word, this recognizer recognizes all possible morphemes contained in the word. Also, this trie contains morpheme forms altered by the morphological rewrite rules. For example it contains *imply* and *impli* too and it recognizes *imply* in both cases. With this technique, it is able to recognize morphemes even if it is altered because of suffixes being attached to it. For example, upon reading *implication*, it recognizes the following morphemes in the following (end) positions, shown with the rules applied: *im(2), imply(y→i,5) ic(6), ate(-e,8), ion(11), on(13).*

- **Chaining of morphemes:** After recognizing the possible morphemes, the algorithm tries to chain the possible morphemes to a sequence that reconstructs the whole word. In English, morphemes can be prefixes, suffixes and stems. The chain recognizer accepts the following regular expression: `prefix* stem+ suffix*`, that is, one or more stems preceded by zero or more prefixes and followed by one or more suffixes. As an exception, if the stem is a regular form of a word, than no additional suffixes can be attached. Also, inflectional suffixes must be attached at the end of the word, and no other suffixes may follow. Additionally, the possible segmentations are filtered by the applicability of the affix rules to match part-of-speech of the lemma and derived word. For example, take the word *buildings*. It segments to *build-ing-s*. *Build* is a verb, and *ing* may be attached to a verb, and it either creates a noun or a continuous verb. If *building* is considered a continuous verb, then no other suffixes may be attached, since *ing* is inflectional in this sense. However, when *building* is considered a noun, then *s* may be attached as a plural marker (note, that it could also be attached as a third person marker to a verb).

- **Enumeration of all legal interpretations:** Unfortunately, legality of suffix attachment may not be decided by linearly processing the morpheme sequence, because prefixes may also modify the part-of-speech of the word. As an example, take the word *enlargement*. *Large* is an adjective, and *ment* can be attached to verbs to create a noun. However, the prefix *en* creates a verb from an adjective, thus *ment* may be attached to *enlarge*. For this reason, the chaining steps enumerates segmentations that have the possibility to result in a legal interpretation, and a further step checks legality thoroughly.

  As an other source of ambiguity, the morphemes of the same segmentation may be ordered in more than one ways to result in legal interpretations but with different meaning. Take for example *undoable*. It can be interpreted as *undo-able* as some action that can be undone. Alternatively, it may

mean *un-doable*, as something that can not be done. Because of this, all possible orderings of prefix and suffix attachments must be enumerated and checked for legality.

5.1.9 Software

In this section we list all the software we have developed or utilized so far. There is one section for software we have developed, one for the software we have incorporated into our system. For the software we have incorporated we also give their function in the system.

Software we have developed

Word graph visualization GUI: We have updated our word graph visualization GUI developed in our previous project to support a richer graph format. The graph visualization tool utilizes JGraph, a free graph visualization library for Java. The updates include proper visualization of directed and labelled arcs and the aesthetic layout and visibility of large graphs. As for the layout, we have utilized a commercial level layout package for JGraph that is free for academic usage.

Ontology Extractor: The ontology extractor performs different tasks (Figure 5.2) for different types of relations incorporated into the ontology: dependency parsing for the dependency relations, coreference resolution for coreference relations, morphological analysis for the morphological relations. In addition we incorporate the semantic relations from Wordnet. Coreference or anaphora resolution is used also to resolve pronominal coreference or anaphora and substitute the antecedent for the anaphor.

We mainly use open source software to realise these tasks. Figure 5.3 is a schematic of the ontology extractor that shows the software used and the connection points, i.e. the points one software gets its input from the output of another software. The list of these software along with their function can be found in Sec. 5.1.9.

NIPG Morphological Analyzer: A morphological analyzer with the following capabilities:

- handles prefixes, suffixes and compounds and their combinations without distorting word endings
- can lemmatize words, handles irregulars even in combination with prefixing and compounding

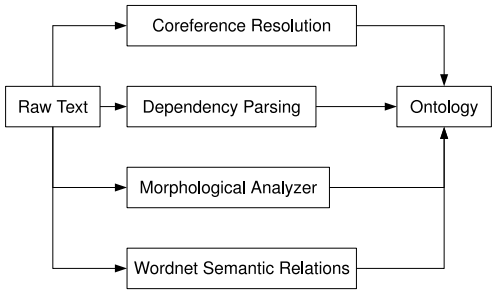


Figure 5.2: **The tasks the ontology extractor performs** The ontology extractor performs different tasks for different types of relations incorporated into the ontology: dependency parsing for the dependency relations, coreference resolution for coreference relations, morphological analysis for the morphological relations. In addition we incorporate the semantic relations from Wordnet. Coreference or anaphora resolution is used also to resolve pronominal coreference or anaphora, and substitute the antecedent for the anaphor in the ontology.

- determines case and tense markers even for irregulars and closed class words (e.g. pronouns)
- can enumerate multiple possible analyses of a word, also taking the desired Part of Speech into account
- provide hypothetical analysis for unknown words (e.g. *un-trool-ed*)

Moreover, the implementation is relatively fast, it analyzes about 20.000 words per second on current computers, and consumes low memory.

Its details can be found in Sec. 5.1.8.

Software we have incorporated into our system

Charniak Parser: Is a phrase structure parser.
Homepage: http://www.cs.brown.edu/ ec/
Function: JavaRap uses this parser to obtain the phrase structure trees of the

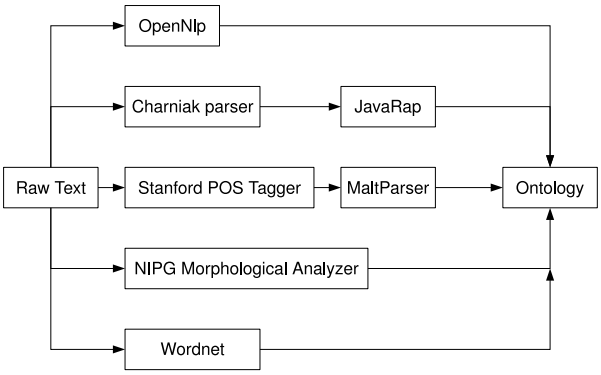


Figure 5.3: **The software used in the ontology extractor** In the boxes are the names of the software used, and the arrow points represent the flow of data between the software. They point from the software that gives its output to the software that gets it as input.

input sentences. It then processes these trees to resolve anaphors.

JavaRap: JavaRap is an anaphora resolution system based on heuristics written in java.
Homepage: http://www.comp.nus.edu.sg/ qiul/NLPTools/JavaRAP.html
Function: We use it to perform anaphora resolution, and substitute the antecedents for the resolved pronominal anaphors in our ontology.

MaltParser: MaltParser is a system for data-driven dependency parsing, which can be used to induce a parsing model from treebank data and to parse new data using an induced model. MaltParser can be characterized as a data-driven parser-generator. While a traditional parser-generator constructs a parser given a grammar, a data-driven parser-generator constructs a parser given a treebank. MaltParser is an implementation of inductive dependency parsing, where the syntactic analysis of a sentence amounts to the derivation of a dependency structure, and where inductive machine learning is used to guide the parser at nondeterministic choice points.
Homepage: http://w3.msi.vxu.se/ jha/maltparser/
Function: We use it to obtain dependency relations for our ontology.

OpenNLP: A collection of natural language processing tools that includes a sentence detector, tokenizer, pos-tagger, shallow and full syntactic parser, and a named-entity detector. It is also capable of coreference resolution based on machine learning.
Homepage: http://opennlp.sourceforge.net/
Function: We use it to perform coreference resolution, and then incorporate the coreference relations into our ontology. We also use it to substitute resolved pronominal anaphors for their antecedents.

Stanford Log-linear Part-Of-Speech Tagger: A part-of-speech tagger. It also contains a sentence splitter and a tokenizer.
Homepage: http://nlp.stanford.edu/software/tagger.shtml
Function: We use it to split the raw input text into tokenized sentences, then assign part-of-speech tags to the words. The output is then feeded into MaltParser.

WordNet: WordNet is a large lexical database of English. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. Synsets are interlinked by means of conceptual-semantic and lexical relations. The resulting network of meaningfully related words and concepts can be navigated with the browser. WordNet is also freely and publicly available for download. WordNet's structure makes it a useful tool for computational linguistics and natural language processing.
Homepage: http://wordnet.princeton.edu/
Function: We incorporate various semantic relationships from WordNet into our ontology.

5.2 Keyword extraction based on word graphs

We have mentioned in the previous report, our approach to keyword extraction builds on our word graph based representation of documents, and we were planning to utilize diffusion or relaxation based methods that simulate activation spreading on the word graph to determine the importance of words. These algorithms are essentially the mathematical implementations of the idea of the *contextual support* [12].

In our approach to keyword extraction, a word is important if it is *supported* by its context. Consider a document that is about *artificial neural networks* for example. If the document mentions a lot of terms related to artificial neural networks then the words *neural* and *network* will be considered important words in the document, even if they do not occur very often. As an opposing example, if a word occurs quite frequently in a document, but is not related to the main topic of the document (for example, 'let us *denote...*' occurs often in mathematical texts, yet *denote* is rarely a keyword), it should not be treated as an important word, since it does not receive support from its context. Thus, we conjecture that it is exactly the graph structure of word-to-word relations that can be used

to decide about the importance of the word.

5.2.1 Algorithm details: 1-dimensional embedding

This term, we have elaborated these ideas and developed a new keyword extraction algorithm. We formalized the idea of contextual support the following way. Let us suppose, that we have n words in a document, and a context graph G contains edge weights $\{g_{ij}\}_1^n$ describing the strength of relationships between words.[1] We wish to calculate a weight vector $\mathbf{w}$ of length n assigning each word a weight w_i proportional to its importance in the document. Let the weight w_i of the i^{th} word be equal to the support it receives from the words it is connected to, weighted by the strength of the connection:

$$w_i := \sum_{j=1}^{n} g_{ij} w_j \qquad \forall i = 1 \dots n \tag{5.3}$$

Obviously, this assignment of weights can not be calculated explicitly since it is recursively referring to the weights of other words. However, the above equations can be used as an iterative assignment to calculate the required weights. Writing the assignment in matrix form, we get:

$$\mathbf{w}^{t+1} := G\mathbf{w}^t \ , \tag{5.4}$$

We have to ensure that the iteration is convergent. If the matrix G is stochastic, the theory of Markov processes can be used to analyze the above iteration. It is known, that a stochastic (square) matrix describes a Markov process, and if that process is ergodic, the above iteration converges to the unique stationary point $\mathbf{w}^\infty$ of the process, from any initial point $\mathbf{w}^0$. We sill use the above iteration to determine word importance weights.

Creating a stochastic matrix from our graph of word-to-word relationships extracted from a document requires only straightforward normalization. However, ensuring that the produced process is ergodic leaves us with more choices. We have experimented with two methods. One straightforward way to ensure ergodicity is to make the matrix symmetric (which corresponds to neglecting the direction of each edge), if the word graph is connected. This ensures the each node can be reached from each other node.

The above iteration resembles the PageRank algorithm used to calculate the importance of web pages on the Internet. The exact formula of the PageRank algorithm is a slightly modified version of Eq. (5.4), which ensures ergod-

[1] The word *weight* is used in two contexts in the literature. One meaning refers to importance, or relevance e.g., the importance of the word in the document, like in our case, here. The other context is the area of graph theory. In the classical case, one talks about nodes and edges between nodes. Edges between nodes are either present or absent; they take values from set $\{0, 1\}$. The concept of weighted graph is a generalization: the value set is extended to reals, sometimes to a set of reals. In this case, one talks about the weight of an edge, or the adjacency matrix of the nodes. This ambiguity should cause no confusion, because they either refer to the nodes or to the edges of the graph.

icity and also deals with *dangling nodes*:

$$\mathbf{w}^{t+1} := \alpha G\mathbf{w}^t + \big(\alpha \mathbf{a}^{\mathbf{T}}\mathbf{w}^{\mathbf{t}} + (1-\alpha)\big)\mathbf{v} \ , \tag{5.5}$$

where the vector $\mathbf{a}$ has a 1 as its ith entry if the ith word has no outgoing links, and 0 otherwise, the vector $\mathbf{v}$ defines so called 'jump' probabilities, which are uniformly $\frac{1}{n}$ in our case, and α is a weighting factor, $\alpha = 0.85$ in case of the PageRank algorithm (see [38] for details). We have also implemented and tested this version of the algorithm.

It is worth noting that our iterative method does not take any initial word weights into account, since it converges to the same stationary point, regardless of the initial weight vector. Previously, we were planning to incorporate word frequencies into the keyword extraction algorithm, but it can be missed in the present form of the iteration. This is so, because the frequencies of the words are implicitly present in the iteration through the frequencies (weights) of their links; it is roughly true, that the frequency of a word equals the sum of the frequencies of its links. We are still using the (normalized version of the) word frequency vector to initialize the iteration (i.e., $\mathbf{w}^0$), because it may decrease the time of convergence.

To sum up, our keyword extraction algorithm works the following way. First, we generate a word graph from a given document using our software developed in the previous term. The result is a graph containing word-to-word morphological, syntactic and semantic relations. The relations have their frequencies assigned to them, and the words also have their frequencies assigned as an initial weight. Then, we apply some graph transformation methods that calculate final edge weights by dealing with multiple edges between words, and ensuring that the resulting matrix is ergodic and stochastic. In this step, we have experimented with a number of transformation methods, the details are listed in Appendix A. (Sec. 5.2.3). Also, we take the logarithm of word frequencies and then normalize them. After this, we run the iterative procedure that results in the relaxed word weights, which we interpret as importance weights. The words receiving the most weight will be considered the keywords of the document.

Our algorithm requires (i) *no additional information about the structure of the document* and (ii) *no training on particular document sets* belonging to a topic. We compare it to the state-of-the-art method both on long and on short documents in the next subsection.

5.2.2 Evaluation results

To test our method for keyword extraction, we have chosen to evaluate it on documents that have keywords assigned to them by their authors. We have downloaded 150 such documents from the Behavioral and Brain Sciences (BBS) online archive[2], and evaluated our method not only against the author supplied keyword set, but also against a standard, state-of-the art keyphrase extraction method, the Keyphrase Extraction Algorithm (KEA)[3].

[2] `http://www.bbsonline.org/`

[3] `http://www.nzdl.org/Kea/`

To shortly summarize the operation of KEA, it learns a decision surface in a supervised manner, based on features of keyphrases in training documents in a given domain or topic. The features it uses include some which build on the structure of documents such as the position of the first occurrence of a word (for example, in scientific texts, keywords are mentioned at the beginning of the document with high probability), and also on other documents in the same domain, like Term Frequency $\times$ Inverse Document Frequency (TF $\times$ IDF) of a phrase (i.e., words that occur in only one document many times, may be an important word in that document).

Thus, KEA (i) requires topic specific documents, because it needs TF and IDF data and (ii) information about the structure of the document, because it assumes that relevant words occur early in (scientific) documents. We can not apply either of these constraints.

To test these properties of our system against KEA, we have used two document sets, the full BBS documents as long documents, and the abstracts of the BBS documents as short documents. Our diffusion based system was tested in two modes. We used the *baseline diffusion model*, i.e., Eq. (5.4) on an undirected, connected word graph (to ensure ergodicity). Results are denoted by Diff-BL. We also used the PageRank equation (Eq. (5.5)) that we denoted by Diff-PR.

Evaluation criteria

To quantify our results, we have used standard measures of precision (P), recall (R), and their combination into an F-measure (F). They are respectively defined as

$$\mathrm{P} = \frac{\text{true positive}}{\text{true positive} + \text{false positive}} \tag{5.6}$$

$$\mathrm{R} = \frac{\text{true positive}}{\text{true positive} + \text{false negative}} \tag{5.7}$$

$$F = \frac{2PR}{P+R} \tag{5.8}$$

The meanings of precision (P) and recall (R) are illustrated in Fig. 5.4. F-measure expresses the balance between the precision and recall by their harmonic mean.

Both KEA and our algorithm assign weights to words and we used the distribution of these weights to quantify the results of the two algorithms against the manually supplied keywords. The general idea is to check how much weight is assigned to those words that are marked as keywords by the authors. It may be important to note that in some cases, the author's keywords do not even occur in the document, because keywords are pointers that may refer to topics related to the thesis of the paper.

Let K_D denote the set of author assigned keywords to a document D, and let $x \in D$ and $x \in K_D$ denote that a word x is contained in a document D or in the set of keywords K_D, respectively. Furthermore, let $|K_D|$ mean the number

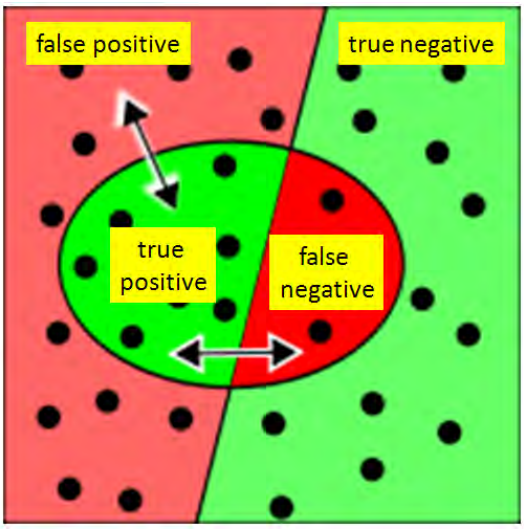


Figure 5.4: Precision and Recall depend on the outcome – separated by the oval curve – of a query and its relation to all relevant documents (the left hand side of the true decision surface) and the non-relevant documents (the right hand side of the true decision surface). Correct results are denoted by green. The more correct results (green), the better. Figure is modified from `http://en.wikipedia.org/wiki/File:Recall-precision.svg`. Copyright granted for any purpose, unless it is against the law.

of keywords assigned by the author to document D. Let $0 \leq w(x) \leq 1$ denote the weight assigned to word x by a keyword extraction algorithm. Then, the definitions of precision P, and recall R for the case of the keyword extractor can be generalized as:

$$P = \frac{\sum_{x \in K_D} w(x)}{\sum_{y \in D} w(y)} \tag{5.9}$$

$$R = \frac{\sum_{x \in K_D} w(x)}{|K_D|} \tag{5.10}$$

The corresponding F-measure remains $F = \frac{2PR}{P+R}$.

Results

The following table details the results for the various cases. The left hand side of the table shows the results (precision, recall and F-score) for full BBS documents, the right side shows the same for the abstracts. The lines correspond to the algorithms discussed above (KEA, Diff-BL and Diff-PR).

As can be seen from the results, our algorithm produces as good as or better word weights as KEA, especially when tested on abstracts and when KEA is

	Full documens			Abstracts		
Algorithm	Prec.	Recall	F-score	Prec.	Recall	F-score
KEA-BBS	0.193	0.325	0.242	0.154	0.190	0.170
Diff-PR	0.141	0.236	0.177	0.167	0.184	0.175
Diff-BL preprocessed	0.303	0.183	0.228	0.250	0.166	0.200
Diff-PR preprocessed	0.243	0.206	0.223	0.179	0.206	0.192

Table 5.2: **Keyword extraction results:** The columns show the precision, recall and F-score of the algorithms run on the full BBS documents and on the abstracts. The algorithms shown are KEA; Diff-PR: diffusion algorithm using PageRank (Eq. (5.5)); Diff-BL preprocessed: baseline diffusion equation (Eq. (5.4)) on a preprocessed graph; Diff-PR preprocessed: diffusion algorithm using PageRank (Eq. (5.5)) on a preprocessed graph. The details of graph preprocessing can be found in Sec. 5.2.3).

not trained in the documents of the topic at hand.

5.2.3 Preprocessing the word graph

To test the effects of different graph preprocessing mechanisms on keyword extraction, we have developed a modular system, where the different modules do different transformations on our graph. The modules can be executed sequentially, in any order. For example, one module combines the different types of edges between two nodes into a single edge by adding them, another module takes all the edges in a graph and takes their logarithms. So if we apply them sequentially, we get a graph in which the weight of an edge between two nodes is the logarithm of the sum of the weights of the edges between the nodes of the original graph. We call these modules graph transmuters.

Our test procedure was the following. First, we applied our transmuters, then ran the PageRank algorithm, calculated the F-scores (for details see Section 5.2.2), and sorted the results (transmuter - F-score pairs) in decreasing order. Then we repeated this procedure for all the 2-long sequential combinations of transmuters, for the 3-long combinations, etc. So, in the end we had the F-measure for all the possible combinations in decreasing order in a table. Looking at this table we could infer the best graph preprocessing applicable to the different cases. We found that going above three long combinations is unnecessary, so we only include the best transmuters up to three long combinations.

We used the following transmuters (they can be found in the attached software: java package nipg.nlp.ontology.transmuters):

- BinaryGraphTransmuter: sets all of the weights of the graph to 1
- CombineByAddingGraphTransmuter: for all pairs of nodes, combines the egdes between them into a single edge by adding their weights
- CombineByMaxingGraphTransmuter: for all pairs of nodes, combines the

egdes between them into a single edge by taking the maximum of their weights

- CombineByMultiplyingGraphTransmuter: for all pairs of nodes, combines the egdes between them into a single edge by multiplying their weights
- GlobalNormalizeGraphTransmuter: divides all the weights with the largest weight separately in each relation category.
- LogGraphTransmuter: takes $\ln(1 + w(e))$ for all edges e in the graph, where $w(e)$ is the weight of the edge
- MorphologyWordnetToAverageGraphTransmuter: Sets the weight of all the wordnet and morphological edges to the average of the weight of the dependency edges
- NormalizeInGraphTransmuter: For all nodes, the length of the vector containing the weights of the edges coming into that node will be 1
- NormalizeOutGraphTransmuter: For all nodes, the length of the vector containing the weights of the edges going out of that node will be 1
- StochasticizeInGraphTransmuter: For all nodes, the sum of the weights of the edges coming into that node will be 1
- StochasticizeOutGraphTransmuter: For all nodes, the sum of the weights of the edges going out of that node will be 1
- SymmetrizeByAddingGraphTransmuter: Creates an undirected graph by duplicating all edges in the opposite direction. If there is already an edge with the same label in the opposing direction, the weight of the new undirected edge will be the sum of the two directed edges.
- SymmetrizeByContextInOutGraphTransmuter: Create a symmetric word similarity graph based on the principle that similar words have similar context using the cosine similarity measure. Takes into account both inbound and outbound links.
- SymmetrizeByContextOutGraphTransmuter: Create a symmetric word similarity graph based on the principle that similar words have similar context using the cosine similarity measure. Takes into account only outbound links.
- SymmetrizeByMaxingGraphTransmuter: Creates an undirected graph by duplicating all edges in the opposite direction. If there is already an edge with the same label in the opposing direction, the weight of the new undirected edge will be the sum of the two directed edges.

Diff-BL	
Without preprocessing	0.05
SymmetrizeByContextInOut	0.19
SymmetrizeByMaxing SymmetrizeByMultiplying	0.24
SymmetrizeByMaxing SymmetrizeByMultiplying NormalizeOut	0.24

Diff-PR	
Without preprocessing	0.17
SymmetrizeByContextOut	0.2
SymmetrizeByMaxing SymmetrizeByMultiplying	0.22
SymmetrizeByMaxing SymmetrizeByMultiplying SymmetrizeByMultiplying	0.23

Table 5.3: **Results of combinatorial graph preprocessing on BBS full documents:** The first column shows the sequentially applied transmuters in order, the second column contains the corresponding F-scores. The rows show the best F-score obtained with no preprocessing, one transmuter applied, two transmuters applied and with three transmuters applied, respectively. The first table was generated using the Diff-BL, the second table by the Diff-PR algorithm (see Section 5.2.1).

- SymmetrizeByMultiplyingGraphTransmuter: Creates an undirected graph by duplicating all edges in the opposite direction. If there is already an edge with the same label in the opposing direction, the weight of the new undirected edge will be the sum of the two directed edges.

We tested the effects of graph preprocessing on the results of keyword extraction for both the Diff-BL and the Diff-PR algorithms described in Section 5.2.1, and for both the BBS abstracts and the BBS full documents. We found significant improvements with preprocessing in all the four cases.

Diff-BL	
Without preprocessing	0.15
SymmetrizeByContextOut	0.17
SymmetrizeByAdding SymmetrizeByContextInOut	0.19
SymmetrizeByMaxing SymmetrizeByMultiplying SymmetrizeByMultiplying	0.2

Diff-PR	
Without preprocessing	0.18
SymmetrizeByContextOut	0.19
SymmetrizeByContextOut SymmetrizeByAdding	0.19
SymmetrizeByAdding SymmetrizeByContextInOut SymmetrizeByMultiplying	0.2

Table 5.4: **Results of combinatorial graph preprocessing on BBS abstracts:** The first column shows the sequentially applied transmuters in order, the second column contains the corresponding F-scores. The rows show the best F-score obtained with no preprocessing, one transmuter applied, two transmuters applied and with three transmuters applied, respectively. The first table was generated using the Diff-BL, the second table by the Diff-PR algorithm (see Section 5.2.1).

5.3 Document similarity and blog diffusion

In order to identify groups of similar blog entries, that is, to identify blog entry diffusion across blogs, first we need means to decide how similar blog entries are. Second, we need methods to find groups (or clusters) of blog entries which contain highly similar blog entries. To test our methods, we have created an experimental test bed, in which we collected blog entries of multiple groups of related documents, along with unrelated, but slightly similar documents, and tested how much our methods are able to find those related groups. We used KurzweilAI network[4] and selected spreading information by hand. We also used Google, to find similar documents. First, we describe our document similarity methods (Subsection 5.3.1). Then we elaborate on our database and present our results (Subsection 5.3.2). Finally, we show, how to pull out groups of related documents without prior knowledge (Subsection 5.3.3).

5.3.1 Document similarity measures

To compare documents, we also rely on the graph representation that we extract using our linguistic tools. We experimented with two kind of similarity measures.

Feature vector based similarity

The first is a feature-vector based similarity, where the features of a document are its words (note, that only open class words (nouns, verbs, adjectives and adverbs) are contained in the word graph of a document, that is, stop words are filtered based on grammatical category). In this representation, each feature (word) is weighted by the output of our diffusion based keyword extraction algorithm. After producing a feature vectors $\mathbf{u}$ and $\mathbf{v}$ for two documents, we have a couple of popular and thoroughly studied choices to calculate a similarity measure between documents. The first is the *cosine angle* of two feature vectors:

$$S_C(\mathbf{u},\mathbf{v}) = \frac{\sum_{i=1}^{n} u_i v_i}{\sqrt{\sum_{i=1}^{n} u_i^2}\sqrt{\sum_{i=1}^{n} v_i^2}} \tag{5.11}$$

The second is the *Jaccard similarity* coefficient:

$$S_J(\mathbf{u},\mathbf{v}) = \frac{\sum_{i=1}^{n} \min(u_i, v_i)}{\sum_{i=1}^{n} \max(u_i, v_i)} \tag{5.12}$$

And the third we have experimented with is the *Dice similarity* coefficient:

$$S_D(\mathbf{u},\mathbf{v}) = \frac{\sum_{i=1}^{n} 2\min(u_i, v_i)}{\sum_{i=1}^{n} u_i + v_i} \tag{5.13}$$

All of these measures produce similarity values in $[0, 1]$. Note that we are throwing away all edge related information from the document graphs when using feature vector based similarity measures, because the feature vectors themselves

[4] `http://www.kurzweilai.net/index.html?flash=1`

(the word weights) were calculated based on the document graph by our novel keyword extraction algorithm.

Graph kernel based similarity

The second method for document comparison is based on *graph kernels*, a more sophisticated method for comparing graphs in general. This method is related to our more-or-less ad-hoc idea for graph comparison that we presented in our previous EOARD project (FA8655-03-1-3036). There, we compared word graphs by counting common nodes and edges. Here, the method also builds upon common nodes and edges, but it takes advantage of the flexibility of the kernel concept.

Kernels extend the idea of *scalar product* to spaces of structured objects, including strings, trees and graphs. *Random walk graph kernels* [6, 61] that we are using here have the intuitive interpretation of preforming random walks on the graphs to be compared and counting the number of *common walks*. This is done by performing random walks on a *direct product graph*. It is worth noting that diffusion equations and random walks are closely related in Euclidean spaces. They are also related for the case of graphs.[5]

Given two graphs $G = (V, E)$ and $G' = (V', E')$ with $|V| = n$ and $|V'| = n'$, their direct product $G_\times$ is a graph with vertex set

$$V_\times = \{(v, v') : v \in V, v' \in V'\} ,$$

and edge set

$$E_\times = \{((u, u'), (v, v')) : (u, v) \in E, (u', v') \in E'\} .$$

In other words $G_\times$ is a graph over pairs of vertices from G and G', and two vertices in $G_\times$ are neighbors if and only if the corresponding vertices in G and G' are both neighbors (see Fig. 5.5). The size of the resulting graph is $n_\times = nn'$. If A and A' are adjacency matrices of G and G', then the adjacency matrix of $G_\times$ is $A_\times = A \otimes A'$, where $\otimes$ denotes the Kronecker product of two matrices.

If the edges in the two graphs are weighted by weight matrix W and W', then we can produce the weight matrix $W_\times = W \otimes W'$, by taking their Kronecker product analogously to the adjacency matrix. From now on, we suppose that weight matrices are columns normalized, so that the entries represent transition probabilities between nodes.

If the nodes of the two graphs are labeled, as in the case of words graphs, where the node labels are the words, then the direct product graph can be restricted to contain only nodes that have *identical*, or *similar* labels in the two graphs. Label similarity can be defined by a *node label kernel* (having values in $[0, 1]$), which can be a *synonymity measure* in case of words as node labels. In this case, the weight of an edge in the direct product graph is multiplied by the synonymity value of the labels of both end node pairs.

[5] `http://en.wikipedia.org/wiki/Random_walk`

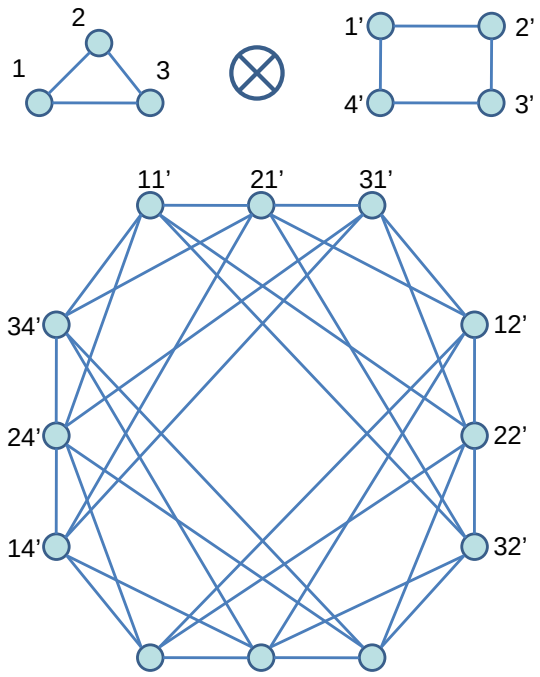


Figure 5.5: Two graphs (top left and right) and their direct product (bottom). Each node of the direct product graph is labelled with a pair of nodes; an edge exists in the direct product if and only if the corresponding nodes are adjacent in both original graphs. For instance, nodes 11' and 32' are adjacent because there is an edge between nodes 1 and 3 in the first, and 1' and 2' in the second graph [61].

Random walk graph kernels calculate similarity of two graphs by simulating random walks on the direct product graph, which corresponds to simulating simultaneous random walks on the two graphs. A well known method to simulate random walks of length k on a graph represented by its adjacency matrix is to take the kth power of the adjacency matrix. Let p and p' be initial node distributions for graphs G and G'. Given the weight matrix $W_\times$, initial probability distribution $p_\times = p \otimes p'$, we can define a kernel on G and G' as (see [6, 61] for details)

$$k(G, G') = \sum_{k=1}^{\infty} \lambda^k W_\times^k p_\times \ . \tag{5.14}$$

The parameter $\lambda < 1$ can be thought of as a discount factor that weights longer walks less than short ones. The initial probability distribution $p_\times$ may be used to weight nodes according to their importance as starting points for the random walks. Usually, a uniform distribution is used here, but in our case, the weight vectors returned by the keyword extraction algorithm may as well be used as

initial vectors.

To evaluate this graph kernel, let us transform the above formula. Suppose, that $\mathbf{I}$ denotes the identity matrix of size $n_\times$, then we have

$$\begin{aligned} k(G,G') &= \sum_{k=1}^{\infty} \lambda^k W_\times^k p_\times \\ &= (\sum_{k=0}^{\infty} \lambda^k W_\times^k - \mathbf{I}) p_\times \\ &= (\mathbf{I} - \lambda W_\times)^{-1} p_\times - p_\times \ . \end{aligned}$$

It can be seen, that the key in evaluating the graph kernel is calculating the inverse $(\mathbf{I} - \lambda W_\times)^{-1}$. Knowing that the size of the matrix to be inverted is $n_\times = nn'$, this becomes computationally intractable for graphs of even moderate size (few hundreds of nodes). Therefore, efficient approximations are required. One simple possibility is to directly compute $x = (\mathbf{I} - \lambda W_\times)^{-1} p_\times$ by means of a fixed-point iteration. Suppose, we are aiming to solve the linear system

$$(\mathbf{I} - \lambda W_\times) x = p_\times \ , \tag{5.15}$$

we can rewrite this equation as

$$x = p_\times + \lambda W_\times x \ . \tag{5.16}$$

Now solving for x is equivalent to finding the fixed point of the above iteration. Letting x_t denote the value of x at iteration t, we set $x_0 = p_\times$, then compute

$$x_{t+1} = p_\times + \lambda W_\times x_t \tag{5.17}$$

repeatedly until $\|x_{t+1} - x_t\| < \varepsilon$, where $\|\cdot\|$ denotes the Euclidean norm, and ε is some predefined tolerance. This is guaranteed to converge if $\lambda < 1/\xi_{max}$, where ξ_{max} is the largest eigenvalue of $W_\times$. In our case, convergence happens for $\lambda < 1$, because of $W_\times$ being a stochastic matrix. In our experiments, we used $\lambda = 0.9$ and $\varepsilon = 0$ because complete convergence happened in all of the cases within few tens of iterations.

Note, that although formula (5.14) satisfies the requirements of a kernel function, it does not produce values between 0 and 1. It produces values greater than 0, returning 0 for graphs that are totally unrelated, which happens when labels are also taken into account and the intersection of the two label sets is empty. In order to receive values in $[0,1]$, we normalize kernel values, generalizing the idea of *cosine distance* of vectors, which is the normalized version of the *scalar product*:

$$\cos(\mathbf{x},\mathbf{y}) = \frac{<\mathbf{x},\mathbf{y}>}{\|\mathbf{x}\| \cdot \|\mathbf{y}\|} = \frac{<\mathbf{x},\mathbf{y}>}{\sqrt{<\mathbf{x},\mathbf{x}><\mathbf{y},\mathbf{y}>}} \ .$$

Analogously, we have

$$\widetilde{k}(G,G') = \frac{k(G,G')}{\sqrt{k(G,G)\,k(G',G')}} \tag{5.18}$$

as our word graph similarity measure.

5.3.2 Evaluation of document similarity

To evaluate our document similarity measures we have collected 130 blog entries organized in the following manner:

1. We have used KurzweilAI[6]. We chose 8 different topics, followed the links of KurzweilAI, and manually collected documents about these news that spread over the Internet, but were rewritten in most cases.

2. In each topic, we entered a list of keywords to a Google query, and the results were scanned manually to organize the results into true and false hits.

3. For each topic we have collected 20 documents; about 10 of them were truly about the topic, and about another 10 were false *hits returned by Google*. In one case, the topic had no false hits, so altogether, we collected 130 documents. Note that we used the main entries of Google that tries to offer different hits about the same topic and organizes similar hits for each entry. Thus, the entries we used are considered (somewhat) different by Google and the differences are similar.

The purpose of this manual collection of a test database was to create a gold standard, against which we can evaluate our document similarity measures and our algorithm that finds related groups of documents.

The document similarity was evaluated the following way. Each document was compared with each other document, which produced a document similarity matrix (or table) of size $N \times N$ supposing we have N documents (N was 130 in this particular study). The matrix contains values between 0 and 1, values close to 1 indicating similar documents, and values close to 0 indicating fairly dissimilar documents. The following table shows a plot of an example similarity matrix, where the values are color coded, (deep) red colors indicating values close to 1, (deep) blue values are close to 0 and yellow values are in between the two.

For the sake of visualization, we organized our documents utilizing our *a priori* knowledge about the manually collected groups: the group of documents belonging to the same Google query formed took compact intervals within the document indices. True and false hits were separated within each interval. Then the similarity matrix should show a *block diagonal* structure, as can be seen in Figure 5.6: for example, documents 10 to 20 are indicated as similar ones, which is right in this case. The above matrix shows about 8 groups of related documents. Note, that the matrix is not fully diagonal, since there are (false) documents that are not part of any group.

This visualization technique is useful for human observation and evaluation of the document similarity measure, but it does not provide a quantitative description of how good a similarity measure is. Of course, an ultimate measure of performance would be to run an algorithm that extracts groups of related

[6] `http://www.kurzweilai.net/index.html?flash=1`

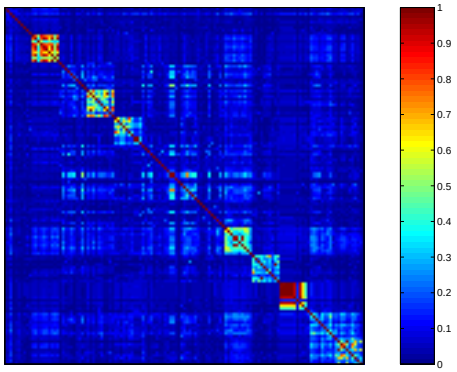


Figure 5.6: **An example document similarity matrix.** Red values indicate high similarity, blue values indicate dissimilarity (see the color bar of the figure). The matrix shows reddish blocks in the diagonal which indicate groups of similar documents.

documents based on the similarity matrix, and check how well it performed compared to our a priory knowledge of related groups. To this end, first we show document similarity measures without reference to any group extraction algorithm. In what follows, we provide two such quantities related to similarity matrices. Later, in Section 5.3.3, we will show results on algorithms aiming to find related groups based on the similarity matrix.

The first is a value of *contrast* defined between the matrix cells that should be 1 and those that should be 0. Again, we are using our *a priory* knowledge that we know which documents are related in this test scenario. Let us denote by $i \sim j$, that document i is similar to document j, and by $i \nsim j$ that document i is dissimilar to document j. Suppose our similarity values are contained in matrix S. Let N_s and N_d denote the number of similar and dissimilar pairs, respectively. Then our contrast C is defined by

$$C(S) = \frac{1}{N_s} \sum_{i \sim j} S_{ij} - \frac{1}{N_d} \sum_{i \nsim j} S_{ij}. \tag{5.19}$$

This contrast is the difference of the mean similarity values of similar and dissimilar document pairs. This contrast value may be a good indicator of how well we will be able to separate groups of related documents from other unrelated documents.

The second quantity is based on the idea that an algorithm that will aim at extracting groups of related documents, will make a *binary* decision at some point: which documents it treats as related and which ones as unrelated. This binary decision making is modelled by thresholding the values in the similarity matrix by some threshold θ, and setting values greater that θ to 1 and others to 0, and counting *true positive, false positive, true negative* and *false negative* hits

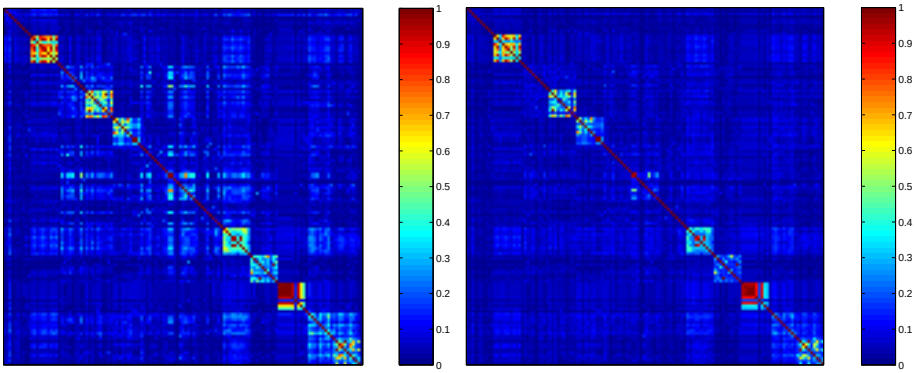
to calculate *precision, recall* and a final *F-measure*. For example, true positives are the values which result in a 1 after thresholding and should be truly a 1 in the matrix, according to our a-priory knowledge. False positives are the values that are thresholded to 1 but should not have been. Negatives are defined similarly. Then the standard definition of precision (P), recall (R) and their harmonic mean F are as given in Eqs. (5.6)-(5.8).

To every threshold value $0 \leq \theta \leq 1$ a single precision value and a single recall value can be calculated. We may then analyze our similarity matrix as a function of θ, for example measure the maximal value of F-measure attainable, or investigate how robust the precision and recall values are against the change of the threshold θ. For example, if a similarity matrix is such that the resulting precision and recall values are robust against a wide range of θ values, it may indicate, that an algorithm that is trying to find related groups will have an more robust, more reliable result with this definition of similarity.

In the following, we list our results with the similarity measures defined in the previous sections and analyze the results according to the above criteria. We list four similarity measures, the three feature-vector based measures (cosine[7], Dice[8], Jaccard[9]) and the graph-kernel based similarity measure. In case of graph kernels, we have experimented with the weight vector produced by keyword extraction. We used it as the initial node distributions instead of a uniform distribution, but it did not produce significant differences, hence those results are not duplicated here. Figure 5.7 shows the resulting document similarity matrices. It can be seen, that the graph-kernel based matrix is quite different from the feature vector based ones; it is a more strict measure, forcing off-diagonal entries to be lower in general and making the block structure more clear, but at the same time, entries in the diagonal also became somewhat lower. Note that *there are 13 groups of documents*, 8 true sets and 5 false sets that we collected.

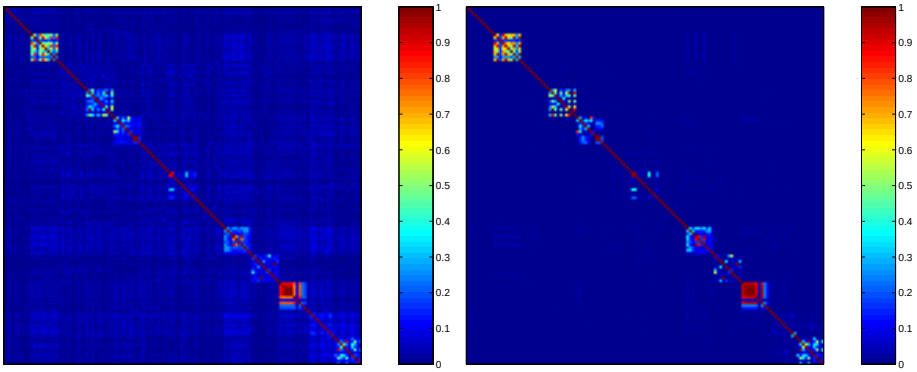
Figure 5.8 shows the change of precision, recall and the F-measure as a function of the threshold θ as introduced above. It can be seen that all similarity measures have a peak F-measure point at some low threshold level as one would expect, and the maximal F-values are around 0.8. The Jaccard and graph-kernel based measures have their maximum at lower threshold levels, because they are stricter than the other two, this can be seen from their matrices; they are cleaner in the off-diagonal entries. At the same time, the cosine measure is the most robust against the value of θ, it produces a slightly flatter F-curve as θ changes. Note, that this kind of robustness correlates with the contrast values of the four similarity matrices: the higher the contrast, the flatter the F-curve.

[7] `http://en.wikipedia.org/wiki/Cosine_similarity`
[8] `http://en.wikipedia.org/wiki/Dice%27s_coefficient`
[9] `http://en.wikipedia.org/wiki/Jaccard_index`



(a) Cosine distance (0.422)

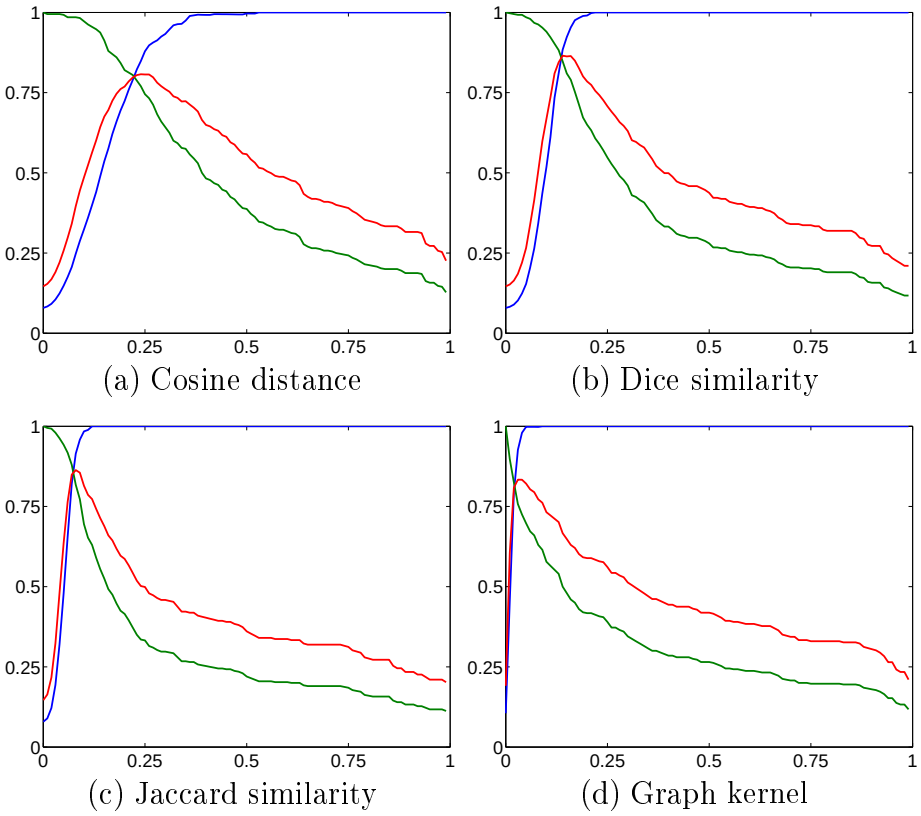
(b) Dice similarity (0.350)



(c) Jaccard similarity (0.286)

(d) Graph kernel (0.314)

Figure 5.7: **Document similarity matrices with four graph similarity measures.** Red values indicate high similarity, blue values indicate dissimilarity. The matrix shows reddish blocks in the diagonal which indicate groups of similar documents. The values in brackets are contrast values defined by (5.19).



(a) Cosine distance (b) Dice similarity

(c) Jaccard similarity (d) Graph kernel

Figure 5.8: **Precision, Recall and F-measure for the four graph similarity measures.** Blue line: precision, green line: recall, red line: F-measure as a function of threshold θ. See text for further details and discussion.

5.3.3 Finding related groups of documents

The document similarity matrices produced by comparison of collected documents can be used to identify groups of related documents. However, this is a nontrivial task. In the previous section, we have used an artificial data set, in which (besides having the a priory knowledge of which documents are related) related documents were carefully ordered, so that the document similarity matrix shows a block diagonal structure. This was merely for the sake of human visualization about the performance of the similarity algorithms. However, in a real data set, we do not know a priory which documents are related, and so it is highly unlikely that a block diagonal matrix arises.

We are looking for state-of-the-art methods that find groups of documents in which the overall similarity is high, and the group cannot be extended by other, highly similar documents. Thinking in graph terms (the similarity values define a similarity graph over the set of documents), an algorithm must look for *densely connected clusters*.

However, traditional clustering algorithms are not sufficient for this purpose, for two reasons. First, they assign each document to a cluster, which is not a good model, since not all documents participate in a group of related blog entries. Second, they assign each document to at most one cluster, that is they are limited to non-overlapping clusters. This feature is not attractive, because in many occasions, a document can be categorized according to two or more different criteria, and so it might be part of more than one group of related documents.

We have two options here. We can look for permutations that exchange indices with the objective that we get the best block-diagonal adjacency matrix. This method uncovers good blocks and the 'coupling' between them, i.e., the documents that belong to more than one block. An extension of this method is to use *all* (or the best) similarity methods and to look for *joint block diagonalization* of the adjacency matrices. This is a novel method and there are fast algorithms for not too large adjacency matrices (see [53,57,58] and references therein) and we are considering to implement it since we are having high F-values and small percentages do count in our project.

Here, we have decided to use a special cluster finding graph algorithm to extract groups of related documents, called Clique Finder, developed by Vicsek et. al. [51]. Also, we have developed and experimented with a simple heuristic method that groups documents based on the similarity matrix, but scales much better than the Clique Finder.

The Clique Finder algorithm

The Clique Finder algorithm works on graphs and aims to find the so called k-*connected components* in graphs. For binary graphs, a k-connected component is a relaxed notion for a clique. While a clique is a subgraph in which all nodes are connected to all other nodes, in a k-connected component, nodes are *connected to at least* k *other nodes* in the subgraph. Furthermore, a k-connected

component is maximal in the sense that no other node can be added to it while retaining k-connectedness. The notion of k-connectedness can be generalized to weighted graphs as well [17], by enforcing an additional requirement to a k-connected set of nodes. This extra requirement states that the *intensity* of a k-connected set of nodes, which is the geometric mean of the edge weights between the nodes in the set, must be above a certain intensity threshold. The Clique Finder algorithm aims to find *all* (weighted) k-connected components in a graph. For details of the algorithm, we refer the interested reader to [51] and [17] and references therein. The algorithm has been successfully applied to various areas, such as extraction of interacting protein groups and analysis of word association networks. Figure 5.9 shows an example graph and its k-connected components.

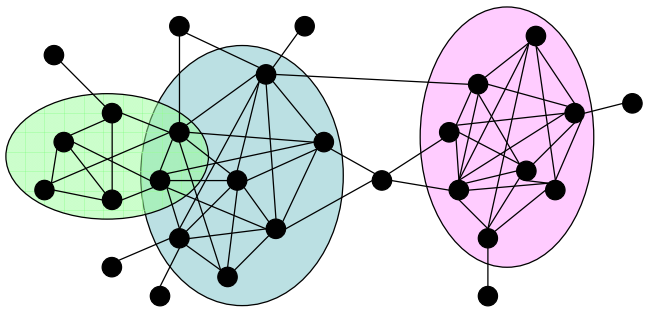


Figure 5.9: **An example for k-connected components in a graph.** The graph has three components shown by the three colors. Those nodes that are not densely connected to a group of other nodes are not contained in any cliques, while some nodes are contained in more than one cliques as well.

We have applied the Clique Finding algorithm to the document graphs resulting from the similarity measures described in the previous section. We tested both the unweighted and the weighted version of the algorithm, on all document similarity graphs introduced in Section 5.3.

For the unweighted algorithm, we had to convert the weighted graph of document similarities to a binary one, which could be done by thresholding the weights and setting the weights below the threshold to 0 and the rest to 1. However, the threshold value can significantly change the resulting graph and so the components found. Also, the algorithm has as additional parameter the value of k in the definition of k-connected components. This value also affects the components found. We have tested a range of parameter values both for the threshold and the value of k. We have evaluated the found components against the a priory known true components in terms of F-measure, just as in Section 5.3.2. In the case of the weighted algorithm, the component intensity threshold is used by the algorithm to prune away components that are too weak, thus we have tested against various values of this parameter in this case. However, as

preprocessing, the graph is still thresholded with a low edge weight threshold and the edges below the threshold are deleted to ensure that it becomes sparse enough for the algorithm to run in a reasonable amount of time. We have used preprocessing weight thresholds low enough to keep the majority of the edges, but high enough to ensure computational tractability. This preprocessing was reinforced by the results, the best intensity values were considerably higher than our thresholding value.

Figure 5.10 shows the results for the four document similarity matrices in the unweighted case, while Figure 5.11 shows the same for the weighted case. It can be seen that in both cases and for all four document similarity measures, there is a range of parameter values, in which the Clique Finding algorithm finds a set of components with fairly good F-measure, around 0.85 being the best for the graph kernel based similarity measure. Note that we have 13 groups of documents with 3 documents that have no false sets and a group of 5 paired sets with both true and false sets. Each set has 10 documents. False sets were collected using Google.

In general, the algorithm finds good components at low values of k, mostly at $k = 3$, however, the optimal threshold varies greatly for the different measures. Although [17] provides a heuristic to find a good intensity threshold value for a given value of k, we believe it is not applicable in our case. The heuristic is based on lowering the intensity threshold value until a huge component starts to emerge, which point is defined to be where the size of the largest component is twice as large as that of the second largest component. The heuristic might work well for scale-free graphs, but in our case there is no reason to assume that the size of the first two largest components differ by a factor of 2. Nonetheless, we have tried the heuristic, but it did really not work for our data, in it general it picked too low threshold values, that is, it found too large components (resulting in poor recall values) as expected.

A heuristic method for finding strongly connected components

We have also experimented with a heuristic method that we developed to find strongly connected components. The method greedily extracts groups of strongly related documents. Its advantage over the Clique Finding algorithm is that it does not require a parameter k, neither an intensity threshold, it scales better and it seems to find good, strongly connected components first and loosely connected ones come at the end. Furthermore, a suitable stopping conditions seems to emerge from our experiments to automatically determine when to stop extracting further components and avoid finding too loosely coupled ones.

The algorithm iteratively finds new strongly connected components. In each iteration it builds a new component, by picking a starting (heaviest/strongest) edge, which defines two starting nodes, and iteratively inserts new nodes to the component. In each step, it inserts the node that is most strongly connected to the current component being built by observing the difference in the intensity of the component and the newly added edges (which we call intensity drop) when trying to insert a given node. This greedy method defines an insertion order

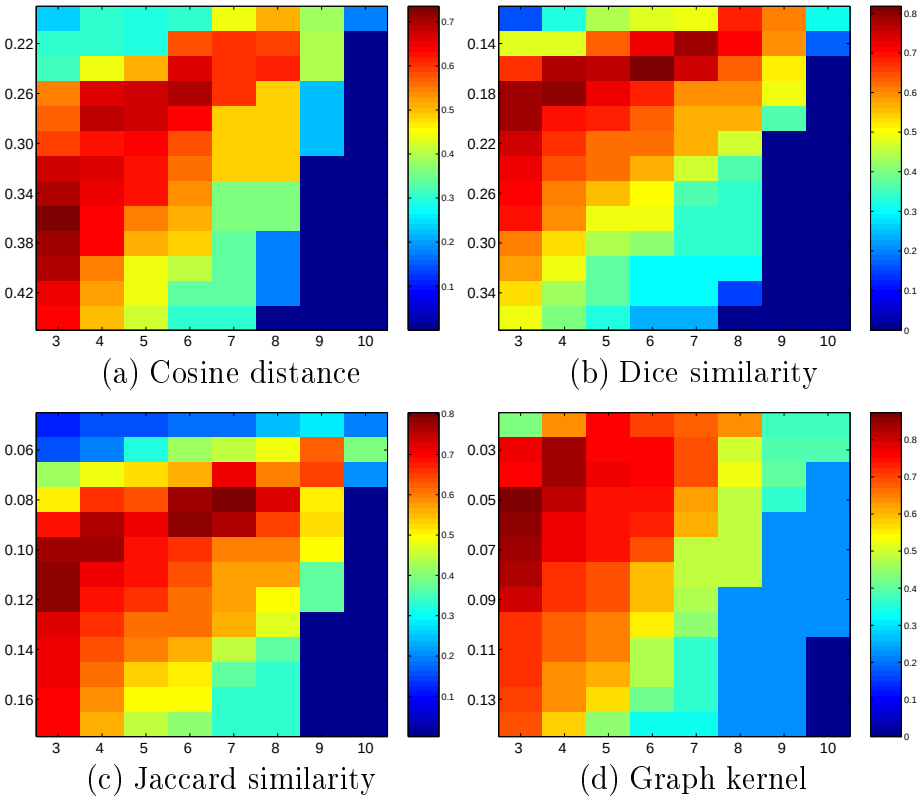
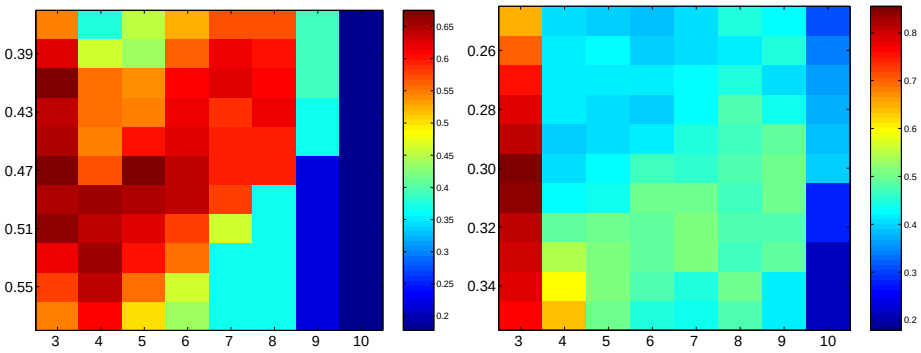
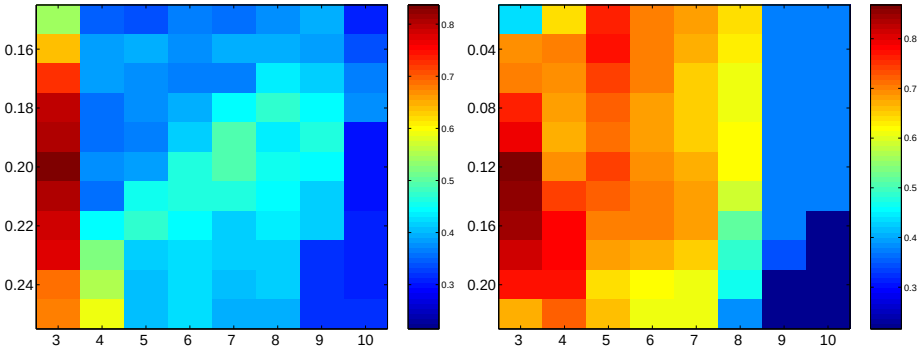


Figure 5.10: **F-measure of components found by the unweighted clique finding algorithm.** Quantitative values of the F-measure are provided by the color bars. Note the differences in the color scales as well as in the y scales. x axis: the value of k; y axis: weight threshold value.

among all nodes to the current component. After this, it finds a cutoff point in this insertion sequence. The cutoff point is characteristic in our database: the intensity drop has a sharp peak (see Fig. 5.12). This indicates strong clustering: upon reaching the 'limits' of the cluster the change of the intensity drops sharply for the initially chosen node. This cutoff point limits the insertion of those nodes that would decrease the intensity of the component strongly, because they are barely connected to the component. When a component is found, its edges are excluded from the selection of an initial node for the next components. There is also an interesting point in the sequence of components found: as the algorithm proceeds, a fairly large component appears; it is considerably larger than the previously found components. This point is chosen to be our stopping criterion for extracting further components (and this large component is not included in the final set of components). More precisely, we chose the point at which the



(a) Cosine distance (0.20) (b) Dice similarity (0.10)



(c) Jaccard similarity (0.05) (d) Graph kernel (0.02)

Figure 5.11: **F-measure of components found by the weighted clique finding algorithm.** Quantitative values of the F-measure are provided by the color bars. Note the differences in the color scales as well as in the y scales. x axis: the value of k; y axis: intensity threshold value. The numbers in the brackets denote the preprocessing weight threshold values used.

size of a newly found component is more than twice as large as any previously found components. This is the same stopping criterion that Farkas et al. [17] are using for finding an optimal intensity threshold in the weighted Clique Finding algorithm. The procedure is detailed in Algorithm 1.

The intensity of a component may be defined in various ways. We have experimented with two measures, namely with the *arithmetic* and with the *geometric mean* of the weights in the component. The geometric mean (as used in the Clique Finding algorithm) is a more strict measure of intensity, it returns zero if one of the edges between any two of the nodes in the component is zero.

We have tested our algorithm by calculating the F-measure of the components found after finding each new component (one iteration in the algorithm). Figure 5.13 shows the results for the four different document similarity measures

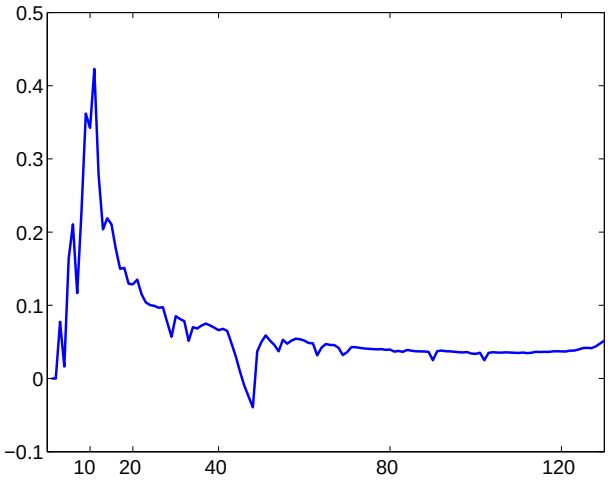


Figure 5.12: **Sharp peak in the intensity drop upon inserting new nodes one-by-one.** x axis: the number of nodes added; y axis: intensity drop.

Algorithm 1 : Find strongly connected components in a graph

inputs: edge weights for the graph

1: **while** size of last component $< 2 \cdot$ size of previous largest component **do**
2: find initial (heaviest) edge for next component
3: calculate the node insertion sequence:
in each step insert the node that decreases
the intensity of the component the least
4: find cutoff point:
the point where the intensity in the insertion sequence drops most
5: exclude the edges of the found component for initial edge selection
6: **end while**

using the *arithmetic mean* of weights as a measure of component intensity.

It can be seen, that the F-measure (blue curves) increases fast at the beginning, reaching a peak of above 0.85 in some cases, but at least about 0.8 at all cases. After reaching a peak, the F-measure drops quickly. These curves indicate, that the algorithm finds truly related sets of documents at the beginning, and later extracts loosely related components as well. At the same time, the size of the components found (green curve) has a sharp increase, which *exactly coincides* with the drop in the F-measure curve. The above mentioned stopping criterion utilizing this sudden increase in component sizes can determine the exact point to stop extracting further (loosely coupled components). For example, with the Dice and Jaccard measures, the algorithm stops after finding components having an F-measure of 0.88 and 0.86 respectively.

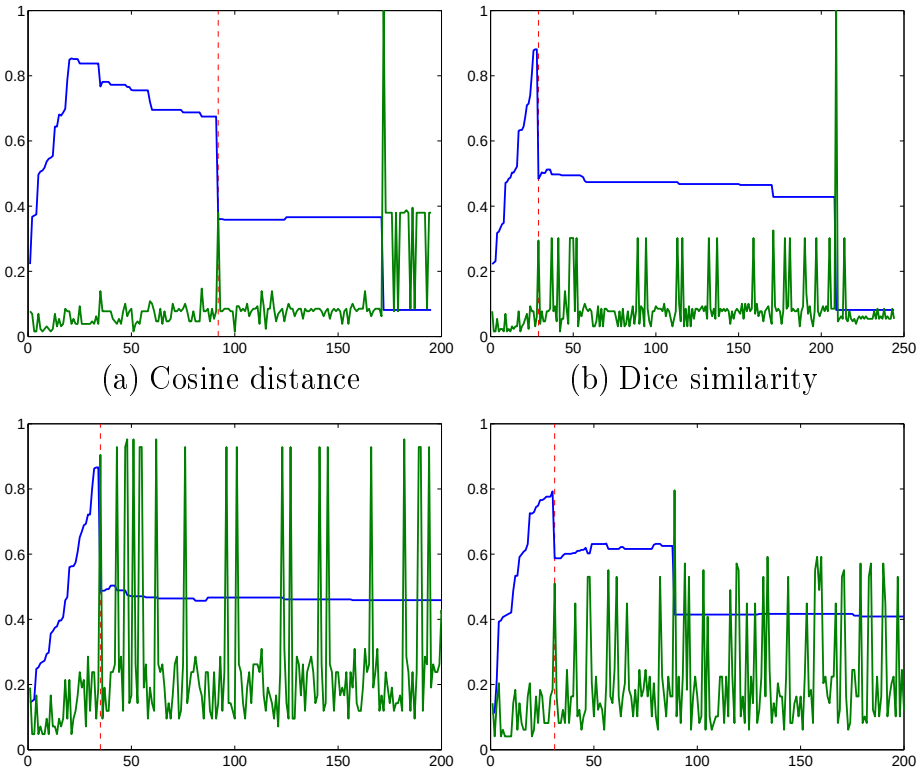
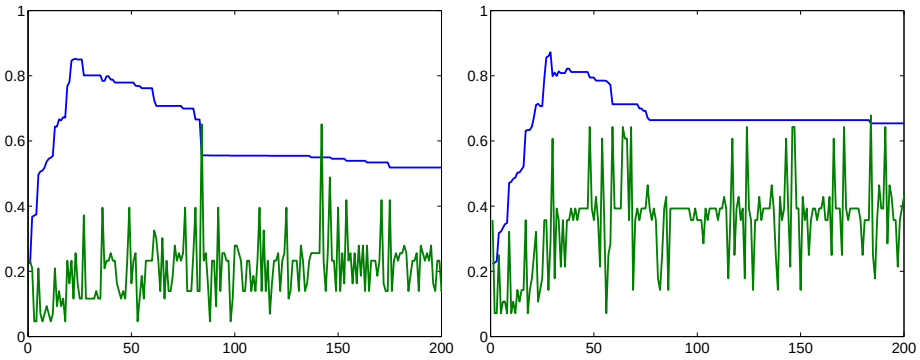


Figure 5.13: **F-measure and size of components found by our heuristic component finding algorithm defining component intensity by arithmetic mean of edge weights.** x axis: the number of components extracted. Components are shown in order, but not all found components are shown. y axis: the F-measure (blue) and size of the components (green) found. The size of the components are normalized for the largest component found. The vertical dotted red line indicates the stopping point selected by our algorithm, which in three cases coincides with the peak point of the F-measure curve.

The use of *geometric mean* of edge weights as component intensity did not perform as good as the *arithmetic mean*. As can be seen from the two examples in Figure 5.14, the F-measure curve has smaller drops after its peak point, as well as the component size curve has smaller increases in it, which changes are not enough before extracting a large number of loosely coupled components as well for our stopping criterion to halt the algorithm early enough.

Our heuristics does not depend on either k, or the *intensity*, but it depends on the peculiar property, the sharp peak of the intensity curve when the algorithm proceeds. *Such a parameter-free algorithm that can reach a*



(a) Cosine distance (b) Dice similarity

Figure 5.14: **F-measure and size of components found by our heuristic component finding algorithm defining component intensity by geometric mean of edge weights.** x axis: the number of components extracted. Components are shown in order, but not all found components are shown. y axis: the F-measure (blue) and size of the components (green) found. The size of the components are normalized for the largest component found.

high F-measure is very relevant for us. Also note, that only edges are excluded when finding new starting points for components, but not later when extending the current component, so the algorithm can find overlapping document clusters and indeed it does in many of the above cases. For example, it often finds a strongly connected core of a true component, and later it finds a broader, but looser version of the same true component. This point deserves further investigations.

5.4 Utilizing synonyms for document similarity

In this section, we detail our results to improve document comparison using background knowledge and accounting for differing vocabularies in documents through synonymity relations among words. First, we define word synonymity measures to automatically extract synonyms from a set of parsed documents, and show the results of that automatic collection. Then, we define some ways of extending document graphs with synonyms, and show our findings on document comparison taking synonyms into account.

5.4.1 Word synonymity measures

As it is well accepted in the field of computational linguistics, the meaning of a word is characterized by the context it is embedded into. We have demonstrated this in our previous work [19], in which we used word contexts defined

by *nearness* in sentences to extract synonymous words. Nearness means that we had counted how often words occurred near each other (within a window of given size) in sentences. Then, the words occurring frequently near a given word made up the context of that word, and we used these contexts as feature vectors to compare words using cosine distance.

In this project, we have used more sophisticated synonymity measures. The contexts of a given word were not defined by the other words that occurred near to it, but rather by the other words that occurred in *grammatical relations* with it. Also, the feature vectors acquired this way were compared with improved, more sophisticated measures.

To introduce the similarity measures, first we need to introduce some notation. We will be dealing with triples of the form (x, r, y), where x and y are words, and r denotes grammatical relations. We denote by $|x, r, y|$ the frequency of word x occurring with word y in relation r. We have collected such frequency information from the British National Corpus (BNC), by grammatical parsing during the last term. We use asterisk notation to denote sets of relations in which one or more arguments run through its ranges. For example $(x, *, y)$ triples denote all relations between x and y, regardless of the exact type of relation r. Analogously, $|x, *, y|$ is the number of such relations. Let $p(x, r, y)$ be the probability of a triple, defined by $p(x, r, y) = |x, r, y|/|*, *, *|$. The probabilities for sets of relations with asterisk notation are defined similarly by normalizing with $|*, *, *|$. Let $T(x)$ denote the set of pairs (r, y) for which $|x, r, y| > 0$.

We have experimented with two measures from the literature. The first is Lin's measure from [42], one of the first papers on the field. Lin used an additional measure $I(x, r, y)$ which is related to the mutual information between x and y and is defined as (see [42] for details)

$$I(x, r, y) = \log \frac{|x, r, y| \times |*, r, *|}{|x, r, *| \times |*, r, y|} \ .$$

Furthermore, define, $\widetilde{T}(x)$ as the set of pairs (r, y) where $I(x, r, y)$ is positive. Then, the similarity between words x and y is defined by

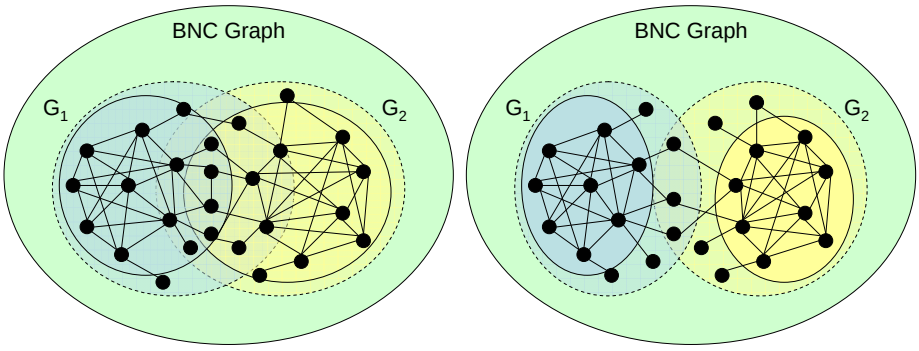
$$sim_{Lin}(x, y) = \frac{\sum_{(r,z)\in\widetilde{T}(x)\cap\widetilde{T}(y)} \big(I(x, r, z) + I(y, r, z)\big)}{\sum_{(r,z)\in\widetilde{T}(x)} I(x, r, z) + \sum_{(r,z)\in\widetilde{T}(y)} I(y, r, z)} \ . \tag{5.20}$$

The second measure is from [13], which paper details and tests a variety of word similarity measures. We have chosen a Jaccard measure based version, because good performance was reported Curran and Moens [13] with this measure:

$$sim_{Jac}(x, y) = \frac{\sum_{(r,z)\in T(x)\cap T(y)} \min\big(W(x, r, z), W(y, r, z)\big)}{\sum_{(r,z)\in T(x)\cup T(y)} \max\big(W(x, r, z), W(y, r, z)\big)} \ , \tag{5.21}$$

where $W(x, r, y)$ is a weight function corresponding to a T-test (see [13] for more details), defined by

$$W(x, r, y) = \frac{p(x, r, y) - p(*, r, y)p(x, *, *)}{\sqrt{p(*, r, y)p(x, *, *)}} \ .$$



(a) Extension of related graphs (b) Extension of unrelated graphs

Figure 5.15: **Extension of graph based on background knowledge.** Related graphs are expected to result in grater increase of overlap, than unrelated. Solid line: original graphs, dotted line: extended graphs. graphs.

The advantage of the Jaccard measure based similarity over Lin's measure is that it takes less computation to evaluate, and gives rise to further speed improvements by approximation through sampling. We have implemented this approximation to enable fast enough synonym generation for our word list of about 50,000 words in the word graph extracted from BNC. In Appendix B, we list a couple of synonyms extracted with these measures.

5.4.2 Extending document graphs with synonyms

In this section we list some novel ways we tried to include background knowledge in document comparison. As we have mentioned in the previous report, we have generated a large word graph by parsing the British National Corpus for grammatical relations, and extended it with morphological and semantic relations from WordNet.[10] We are aiming to use this large graph to extend small graphs extracted from blog entries. The idea is illustrated in Figure 5.15: by extending documents with synonyms and related words, two graphs that are about the same topic will have higher overlap, hence higher similarity. We would expect less increase in overlap for unrelated graphs.

We have devised four methods to extend word graphs with related words or synonyms. Three of these extend the word graph of a document explicitly, and the fourth extends the graphs implicitly during graph comparison in the graph kernel method.

1. Extend the word graph with all *related* words in the BNC graph that are *directly* linked to one of the words in the graph through a grammatical relation, i.e. add the 1 neighbors of the words to the graph.

[10] `http://wordnet.princeton.edu/`

2. Extend the word graph with *top ranked synonyms* of its words, extracted from the BNC graph using the synonymity measures defined in the previous section.

   (a) Connect synonyms to the word they are related to with links labeled as 'synonym', as shown in Figure 5.16 (a). (The text of the label itself does not matter, what matters is where the word is linked.) The weight of the link equals the synonymity value between the two words, multiplied by a factor of 25, which was determined experimentally, to add more emphasis to the relatively weak synonymity links when compared to all other link weights in the graph.

   (b) Link synonyms to the *context* of the word they are related to, by copying its grammatical links. This is justified by the assumption that synonymous words tend to appear in similar contexts. This kind of extension generates identical contexts for two synonymous words in the resulting graph, as shown in Figure 5.16 (b).

   We have also experimented with restricting these kind of extensions to words occurring in the union of the two graphs being compared, i.e. extend the first graph with the synonyms occurring in the second graph and vice versa. In addition, we have further restricted the extension to words that are not too frequent in BNC.

3. Extend graphs implicitly during document comparison with graph kernels, by utilizing word similarity extracted from BNC as *node label kernel* (see Section 5.3.1).

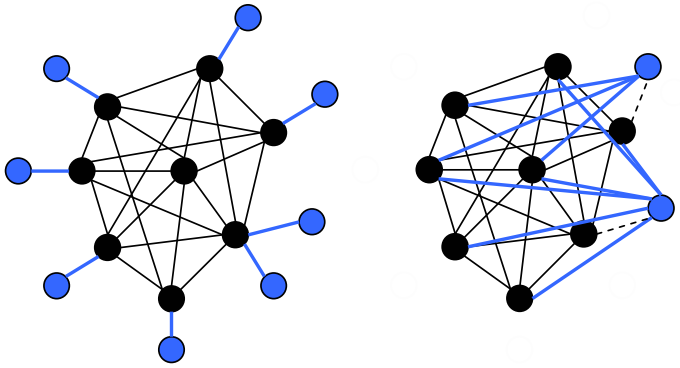
5.4.3 Document similarity on the extended graphs

We have evaluated our document similarity measures defined in Section 5.3 on the document graphs extended with synonyms. The keyword extraction algorithm was run after the extension to allow weighing of newly inserted words as well. After that, document comparison proceeded just as in the unextended case. In case of graph kernel based document similarity, the implicit extension based on node label kernel was also tested.

To test whether extension with synonyms resulted in improvements in document comparison, we examined the resulting document similarity matrices and the corresponding maximal attainable F-measure by choosing the right threshold θ as defined in Section 5.3.2.

Unfortunately, most of our attempts failed in utilizing extension by synonyms in document similarity. Extension in most cases spoiled performance and we could improve the results only by a slight margin and only in a single case. In this respect, we should note, however, that our graph comparison methods are highly efficient and *we worked on improving our already high F values.*

The general lesson that we learned is that we have to perform *extension and restriction* very *carefully.* The only extension method that was able to increase



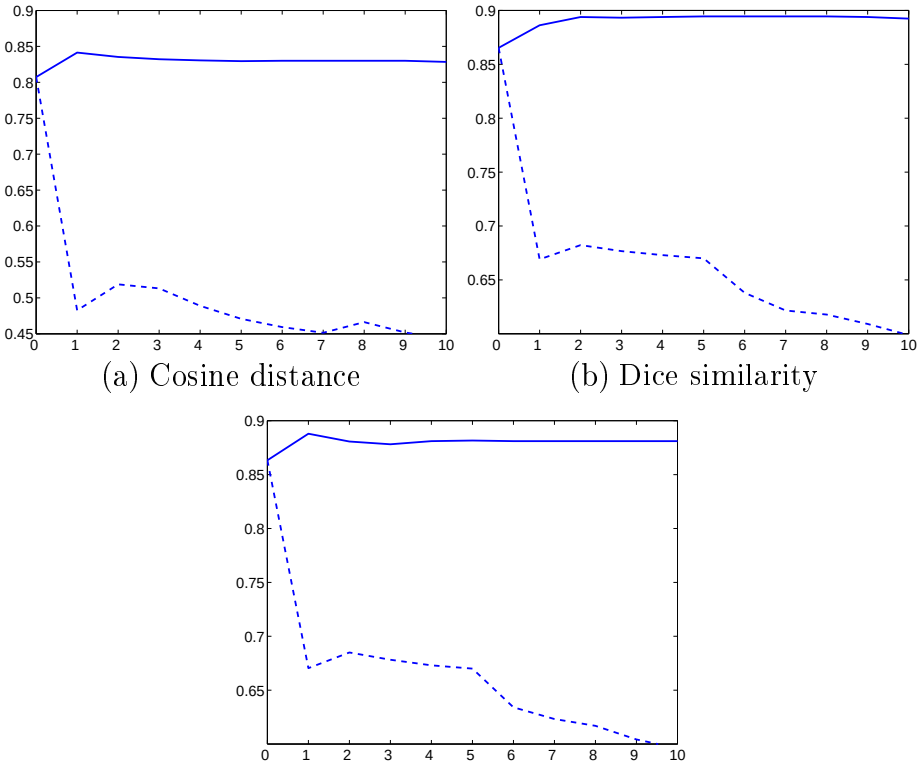
(a) Adding synonym links (b) Copying contextual links

Figure 5.16: **Extension of graphs with synonyms.** (a) synonyms are linked by edges labelled 'synonym' to the original words. (b) synonyms are linked to the context of the original word with the labels of the contextual links copied.

performance was when (i) we used the synonyms list from BNC to extend the graphs, (ii) we restricted the list of synonyms to *words that are not too frequent* in BNC (the extension seems to be quite insensitive to the exact frequency threshold; thresholding with maximum frequency of 50 and 250 produced very similar results), (iii) we *did not* restrict the extension to words occurring in either of the two documents being compared and (iv) the synonyms were linked to their related word in the graphs with a *single link labeled 'synonym'*. Figure 5.17 shows the increase in the maximal attainable F-measure for the successful case of extension, along with an unsuccessful case. Other unsuccessful cases produced similar decrease in the F-measure. The successful extension peaks when 1 or 2 synonyms per word were used, adding more related words did not increase the performance further, but it did not decrease it either, suggesting that the method is somehow robust against being extended with further related but not too frequent words.

In short, if F-values are high, synonymity should be handled very carefully: it is hard to improve the results, whereas it is easy to spoil them. Graph based comparisons are highly efficient, so more sophisticated methods are needed to take advantage of the background knowledge.

We have also examined what exactly increased the F-measure during the extension. Figure 5.18 shows the change in precision, recall and F-measure for the cosine similarity measure (the change in the other measures is similar). It can be seen, that it is the precision that increases around the peak point, and that recall does not change significantly. This seems to suggest that it is the number of false positives hits that decreases, but more detailed investigation should be carried out about this matter.



(a) Cosine distance

(b) Dice similarity

(c) Jaccard similarity

Figure 5.17: **Successful and unsuccessful extension of graphs with synonyms for document comparison.** x axis: the number of synonymous words used in extension; y axis: maximal attainable F-measure. The solid line shows the increase in F-measure for the successful extension method; the dotted line shows the decrease in F-measure for the same method without restricting synonyms to low frequency words in BNC.

5.5 Learning topics from documents

5.5.1 The algorithm

Dictionary learning algorithms learn basis vectors that capture high-level features of unlabeled data. They model each input vector as the linear combination of a few basis vectors. We use sparse coding to break documents down into topics. In our case, the input vectors are Bag of Words representations of documents. Each column of the input matrix $\mathbf{X} = [\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^n] \in \mathbb{R}^{mxn}$ is a document, where there are m distinct words across all documents. The documents are stemmed and the stop words are removed.

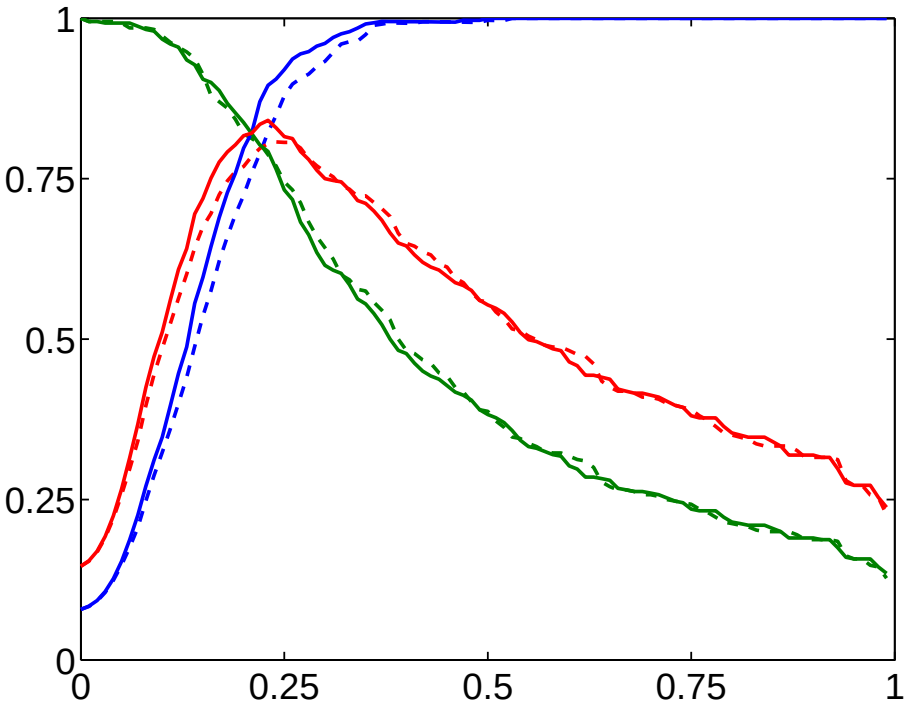


Figure 5.18: **Precision, Recall and F-measure for the cosine graph similarity measure.** Blue line: precision, green line: recall, red line: F-measure as a function of threshold θ. Dotted: before extension with synonyms; solid: after extension.

We want to factorize the matrix $\mathbf{X}$ into two matrices, $\mathbf{D} = [\mathbf{d}^1, \mathbf{d}^2, \ldots, \mathbf{d}^n]$ and $\mathbf{A} = [\boldsymbol{\alpha}^1, \boldsymbol{\alpha}^2, \ldots, \boldsymbol{\alpha}^n]$, where $\mathbf{D}$, the dictionary contains the basis vectors or topics, and $\mathbf{A}$ contains the coefficients in the linear combination of each document: $\mathbf{x}^1 \approx \mathbf{D}*\boldsymbol{\alpha}^1$, $\mathbf{x}^2 \approx \mathbf{D}*\boldsymbol{\alpha}^2$, and so on. In matrix notation: $\mathbf{X} \approx \mathbf{D}*\mathbf{A}$.

Furthermore, we impose a structure on the elements (columns) of the dictionary $\mathbf{D}$. Each $\boldsymbol{d}^i$ is embedded in a structure $\mathcal{G} = \{I_1, I_2, \ldots, I_n\}$, where each $I_i \subseteq \{1, \ldots, n\}]$ represents the neighbors of $\boldsymbol{d}^i$. For example, if $I_1 = \{1, 2, 3\}$, then the first column and the next two are neighbors.

We want to solve

$$f_n(\mathbf{D}) = \frac{1}{n} \sum_{i=1}^{n} l_{\kappa,\eta,\mathcal{G}}(\mathbf{x}_i, \mathbf{D}), \tag{5.22}$$

Where

$$l_{\kappa,\eta,\mathcal{G}}(\mathbf{x}, \mathbf{D}) = \min_{\boldsymbol{\alpha} \in \mathbb{R}^k} \left[\frac{1}{2} \|\mathbf{x} - \mathbf{D}\boldsymbol{\alpha}\|_2^2 + \kappa \Omega_{\mathcal{G},\eta}(\boldsymbol{\alpha}) \right] \quad (\kappa > 0). \tag{5.23}$$

The $\boldsymbol{\Omega}$ regularizer has two purposes, in accord with [31]. It enforces sparsity

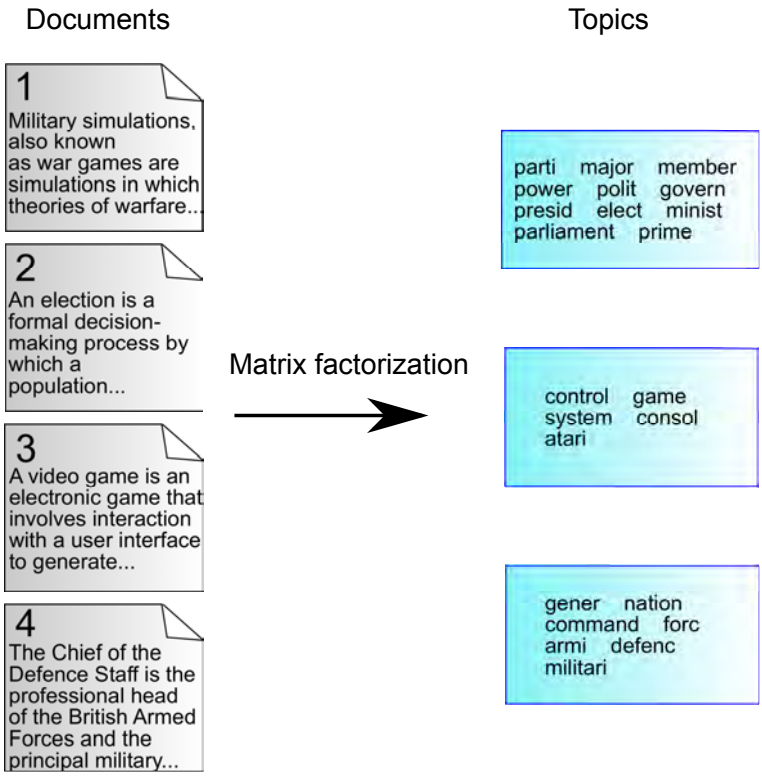


Figure 5.19: **Learning topics from documents** The matrix $\mathbf{X}$ that contains the documents as its columns in a Bag of Words representation is factorized into $\mathbf{X} \approx \mathbf{D} * \mathbf{A}$. As a result, each column of $\mathbf{D}$ will contain a topic. Each column $\boldsymbol{\alpha}^i$ of $\mathbf{A}$ contains the coefficients for topics for the corresponding $\mathbf{x}^i$ document (see also Fig. 5.20). This a hypothetical example.

and the topography the columns of D are embedded in. If $\mathbf{y}_G$ denotes the vector where all the coordinates that are not in the set $G \subseteq \{1, \ldots, n\}$ are set to zero, then

$$\Omega_{\mathcal{G},\eta}(\mathbf{y}) = \|(\|\mathbf{y}_G\|_2)_{G\in\mathcal{G}}\|_\eta = \left[\sum_{G\in\mathcal{G}} (\|\mathbf{y}_G\|_2)^\eta\right]^{\frac{1}{\eta}} \quad \eta \in (0,1] \tag{5.24}$$

If topography is not introduced and thus there are no neighbors, that is, for all $i \in \{1, \ldots, n\}$, $I_i = \{i\}$, and if $\eta = 1$, then the optimization task relaxes to the l_1-norm based Lasso optimization task. If neighborhoods are included and I_is have more than a single element, then the few element choice generalizes to few group choice. However, elements in each group $\boldsymbol{\alpha}_{\boldsymbol{I_i}}$ are subject to the ℓ_2 norm and the selected groups can have dense representations. This allows us to

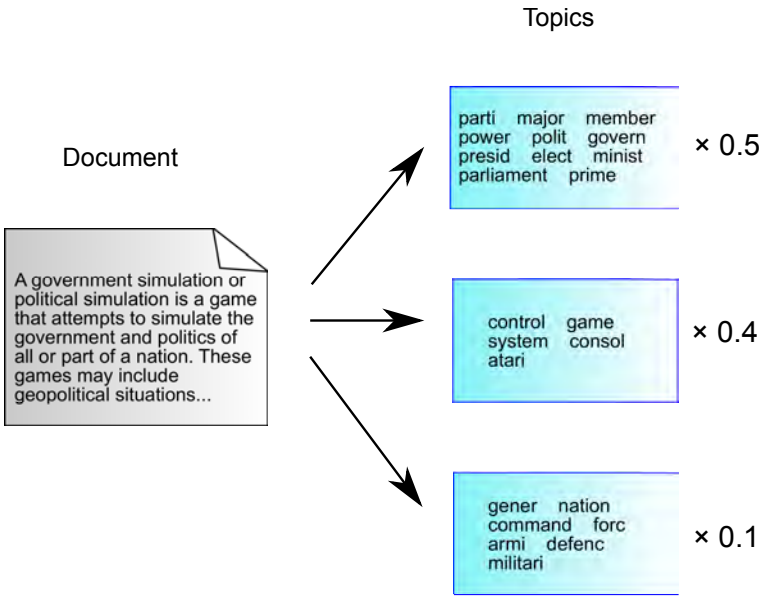


Figure 5.20: **Breaking down a document into topics** A document $\mathbf{x}^i$ is analyzed as a linear combination of topics. $\mathbf{x}^i$ is synthesized as $\mathbf{x}^i \approx \mathbf{D} * \boldsymbol{\alpha}^i$. $\boldsymbol{\alpha}^i$ contains a coefficient for each topic $\mathbf{d}^i$. $\boldsymbol{\alpha}^i_j$ tells us about the extent document i is about topic j. This a hypothetical example.

incorporate prior knowledge about the dictionary elements (every coordinate of α corresponds to exactly one column in D).

The other parameter of $\boldsymbol{\Omega}$ is η. If $\eta = 1$, then we sparsify the α_{I_i} in l_1-norm. Decreasing η leads to more aggressive sparsification.

In matrix notation, our task is to solve the following optimization problem:

$$J_{\lambda,\kappa,\eta,\mathcal{G}}(\mathbf{D},\mathbf{L}) = \left[\frac{1}{2}\left\|\mathbf{X}-\mathbf{DL}\right\|_F^2 + \kappa\Omega_{\mathcal{G},\eta}(\mathbf{L})\right] \to \min_{\mathbf{D}\in\mathcal{C},\mathbf{L}\in\mathbb{R}^{k\times n}}, \tag{5.25}$$

We imposed additional constraints on $\mathbf{D}$ and $\mathbf{A}$. For every element d_{ij} of $\mathbf{D}$, $d_{ij} \geq 0$. Also, for every element α_{ij} of $\mathbf{A}$, $\alpha_{ij} \geq 0$. We represent topics in a mixture of topics model: every word can only contribute positively to a topic, and every topic can only contribute positively to a document. So the problem we are solving is a nonnegative matrix factorization problem. There is also a normalization constraint on $\mathbf{D}$: for every $\mathbf{d}^i$ $(i \in \{1,\ldots,n\})$, $\sum_{j=1}^{m} d_{ij} = 1$.

It is a relevant development in our other project that our iterative learning algorithm is online: it can process the $\mathbf{x}^i$-s one-by-one. This is a tremendous advantage compared to batch algorithms: our memory footprint is decreased tremendously, as we do not have to keep the whole $\mathbf{X}$ matrix in memory.

The algorithm learns $\mathbf{D}$ and $\mathbf{A}$ simultaneously. It is very similar to Algorithm 1 in [43], with the following difference. In one iteration, there are two main steps:

1. We optimize the $\boldsymbol{\alpha}_t$ hidden representation for the current $\mathbf{x}_t$ input using the $\mathbf{D}_t - 1$ obtained in the previous iteration.

$$\boldsymbol{\alpha}_t = arg \min_{\boldsymbol{\alpha} \in \mathbb{R}^k} \left[\frac{1}{2} \|\mathbf{x}_t - \mathbf{D}_{t-1}\boldsymbol{\alpha}\|_2^2 + \kappa \Omega_{\mathcal{G},\eta}(\boldsymbol{\alpha}) \right] \tag{5.26}$$

2. We update $\mathbf{D}_t - 1$ using the previous $\{\boldsymbol{\alpha_i}\}_{i=1,\ldots,t}$

$$\hat{f}_t(\mathbf{D}) = \frac{1}{t} \sum_{i=1}^{t} \left[\frac{1}{2} \|\mathbf{x}_i - \mathbf{D}\boldsymbol{\alpha}_i\|_2^2 + \kappa \Omega_{\mathcal{G},\eta}(\boldsymbol{\alpha}_i) \right] \rightarrow \min_{\mathbf{D} \in \mathcal{C}} \tag{5.27}$$

The algorithm works with mini-batches: it processes k inputs in an iteration, where k is a parameter.

5.5.2 Experiments

We conducted experiments on pages uniformly sampled from Wikipedia. We have experimented with the following topography. The topics are placed on a hexagonal grid on a torus. Every $\boldsymbol{\alpha}_i$ has exactly 6 neighbors (Fig. 5.21.

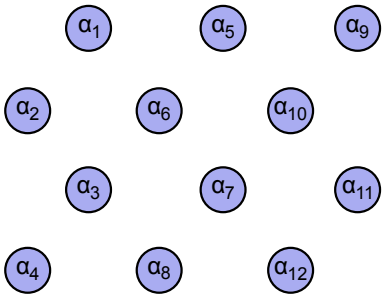


Figure 5.21: **The hexagonal grid the topics are embedded in.** Each coefficient in α represents the contribution of a topic to a document. Each α_i has exactly six neighbors. For example, the neighbors of α_6 are α_1, α_2, α_3, α_7, α_{10}, α_5. The grid is placed in a torus, e.g., α_1 and α_9 are neighbors.

We used 10,000 documents from Wikipedia, and kept all words that occurred at least 10 times in the corpus. The words were stemmed. We placed the topics on a 20 by 20 grid, so the dimension of α was 400. We set κ to 2^{-10} and η to 0.9. The learning rate ρ was 1. The size of the minibatches, k, is 16. Out of the 400 topics, 70 contained more than 5 nonzero elements (or, in our context, words). Table 5.5 contains 40 average-sized, Table 5.6 contains all of the larger topics.

It can be seen even from these preliminary results that all of the topics are meaningful. In the future, we plan to reduce the sparsity of the topics, and also improving their embedding in the topography. Sparsity of the topics could be reduced for example by separating the typical and specific information in the documents using RPCA, and then analyzing the typical information.

These results open up new, exciting possibilities in nearly all aspects of our project. The meaning of texts can be determined with more precision if we know the topic of the text. The dialog system can work much more efficiently if we know the topic of the question. But the most immediate and important application is to analogy detection. It can be made much simpler and more robust, without dependence on the link structures or information cascades.

5.5.3 Detecting analogies based on topics

When we have successfully identified the topics a document is about, we can find analogies in the following way. We look for sets of topics that co-occur frequently, or associations between topics. Identifying these is a well-studied problem in data mining, beginning with [2]. There are efficient algorithms for these tasks. It is natural to connect them with sparse representations, as their input vectors are typically very sparse: 20-30 nonzero elements in vectors of dimension 10^5 – 10^6 is considered typical.

In our problem, we have the $\boldsymbol{\alpha}^i$-s as input vectors (i.e., the transactions), and we are looking for associations between its coordinates (i.e., the item set), or co-occurring sets of coordinates of $\boldsymbol{\alpha}^i$. Each coordinate j of $\boldsymbol{\alpha}^i$ represents the degree in which document i belongs to topic j. So we are looking for topics that co-occur frequently in documents, or association rules between topics.

We can detect the following analogies:

1. Topics that co-occur frequently. Consider an example with two topics. One of the topics could be about an actor, the other about a movie. If these two occur together frequently, it is very likely that the actor acts in the film.

2. Entailment: the presence of one topic frequently implies the presence of another. This is similar to co-occurrence, only not symmetric. Consider the following example. Our corpus involves lots of movies in a longer time period, and if a movie is mentioned, the director is often also mentioned. If there is a mention of the director, there will be no single film that stands out, because there are several films he has directed. In this case, the movie entails one director, but a director entails several movies. We can fill in the consequent (e.g., Who has been involved with the film?), the antecedent (e.g., What films has Mr. X. directed?) or both (e.g., In addition to film Z., what has Mr. X directed?). This is a way to find analogies in connection with a predefined topic.

Using analogies is very similar in all cases. We look for documents in each of the topics of the analogy, then combine them. In the example of co-occurrence,

Topic number	Words in the topic
1	perform includ map galileo project peter eden cartograph evm
2	problem theori mathemat solv mathematician
3	year relat nation pari govern french territori franc
4	william emperor duke frederick prussia
5	separ respons solid filter particl filtrat
6	year time plai game score leagu team player
7	relat includ intern state nation polit govern presid countri
8	control game system consol atari
9	fill color code node algorithm pixel freenet
10	english ndash singer french footbal actress player
11	parti major member power polit govern presid elect minist parliament prime
12	year gener member product oper servic compani telephon
13	time perform includ plai work instrument compos music record song
14	soviet afghanistan kabul afghan khan
15	year birth peopl male femal live popul rate total nation ethnic countri est
16	year time includ intern nation world countri
17	gener nation command forc armi defenc militari
18	year plai home win record leagu club season team
19	flight hijack fbi hanjour mihdhar
20	year includ part nation british centuri island sea south land isl
21	sound modul synthesi signal frequenc carrier harmon hertz
22	research school student univers hall scienc institut colleg
23	form structur bond hydrogen ion molecul coval
24	structur element energi atom chemic moselei hydrogen molecul
25	diseas medic physician ancient centuri medicin galen hippocrat hippocr
26	central pacif gosford salvador hondura
27	year time centuri son king death kingdom henri reign
28	develop inform time gener includ system oper
29	step left danc kelli countri dancer
30	refer form word english mean
31	goal score leagu club footbal cup
32	year time show televis converg charact episod seri releas star sum
33	year time life publish work art
34	classic text chines tao ching
35	templ period greek centuri hera golden egyptian ephesu delphi
36	cycl power effici heat fuel engin jet combust
37	canadian dai compani canada macdonald
38	roman holi christian orthodox cathol church council bishop saint christ
39	protein water reaction fat acid carbon fatti carboxyl ester hydrolysi
40	gener includ base oper state servic war command forc unit navi air fighter missil

Table 5.5: **Average sizes topics**
The topics are is embedded in a 20 by 20 hexagonal grid on a torus. We show 40 topics of average size. It can be seen that all of the topics are meaningful.

Topic number	Words in the topic
1	peopl time cultur histori event centuri great historian histor religion republ herodotu
2	region part river water sea north south land west countri road plain border mountain lake flow indu
3	peopl time number gener form includ group call member subgroup element isomorph homomorph
4	develop control product oper state design servic fly wing unit air isbn flight phantom aircraft engin pilot carrier fighter radar squadron
5	year independ member execut nation court assembl repres power polit govern constitut branch presid vote elect minist parliament council prime legisl
6	refer develop languag peopl number gener form includ word program standard centuri origin dialect spoken linguist speaker vowel
7	form imag contribut onlin energi commerc count microscop electron ion ionic valenc quantum
8	year time live singl perform plai group rock featur member music album record song band releas tour chart guitar
9	inform time program problem network work univers scienc machin code comput softwar algorithm
10	point product set complet analysi space linear launch topolog vector theorem hausdorff tsiolkovski
11	year peopl time popul includ area locat part state nation school univers centuri citi center town
12	refer time number gener line includ factor neg call standard numer set posit imaginari letter prime digit
13	refer peopl time call tradit greek centuri son god goddess human worship jewish christian origin religion jesu faith

Table 5.6: **Larger topics from an experiment on a hexagonal grid**
The topics are is embedded in a 20 by 20 hexagonal grid on a torus. We show all 13 of the larger topics. It can be seen that all of the topics are meaningful.

we do the following. On one hand, we look for documents that contain the topic about the actor. On the other hand, we look for documents about the film. If we find both, we can match them, and ask questions like 'What will be the consequence of canceling the film on the career of the actor?'. The other cases are similar, only we use the obtained documents in a different way.

Obtaining documents with specific topics is very fast, as the topical representation (i.e., the $\boldsymbol{\alpha}^i$-s) of the documents can also be stored in an inverted index using Lucene.

5.6 Learning a topography of topics

In the previous semester, we have successfully broken down documents into topics, but our preliminary results could be improved upon. The topics were largely independent, and they were very sparse. In this semester, we have fine-tuned our algorithm to establish topography between denser topics. The topics are embedded in a topography, where topics that are near each other are more similar than topics that are further apart. In the following, we first revisit the algorithm, then we describe the modifications and the new results.

5.6.1 The original algorithm

Dictionary learning algorithms learn basis vectors that capture high-level features of unlabeled data. They model each input vector as the linear combination of a few basis vectors. We use sparse coding to break documents down into topics. In our case, the input vectors are Bag of Words representations of documents. Each column of the input matrix $\mathbf{X} = [\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^n] \in \mathbb{R}^{mxn}$ is a document, where there are m distinct words across all documents. The documents are stemmed and the stop words are removed.

We want to factorize the matrix $\mathbf{X}$ into two matrices, $\mathbf{D} = [\mathbf{d}^1, \mathbf{d}^2, \ldots, \mathbf{d}^n]$ and $\mathbf{A} = [\boldsymbol{\alpha}^1, \boldsymbol{\alpha}^2, \ldots, \boldsymbol{\alpha}^n]$, where $\mathbf{D}$, the dictionary contains the basis vectors or topics, and $\mathbf{A}$ contains the coefficients in the linear combination of each document: $\mathbf{x}^1 \approx \mathbf{D}*\boldsymbol{\alpha}^1$, $\mathbf{x}^2 \approx \mathbf{D}*\boldsymbol{\alpha}^2$, and so on. In matrix notation: $\mathbf{X} \approx \mathbf{D}*\mathbf{A}$.

Furthermore, we impose a structure on the elements (columns) of the dictionary $\mathbf{D}$. Each $\boldsymbol{d}^i$ is embedded in a structure $\mathcal{G} = \{I_1, I_2, \ldots, I_n\}$, where each $I_i \subseteq \{1, \ldots, n\}]$ represents the neighbors of $\boldsymbol{d}^i$. For example, if $I_1 = \{1, 2, 3\}$, then the first column and the next two are neighbors.

We want to solve

$$f_n(\mathbf{D}) = \frac{1}{n} \sum_{i=1}^{n} l_{\kappa,\eta,\mathcal{G}}(\mathbf{x}_i, \mathbf{D}), \tag{5.28}$$

Where

$$l_{\kappa,\eta,\mathcal{G}}(\mathbf{x}, \mathbf{D}) = \min_{\boldsymbol{\alpha} \in \mathbb{R}^k} \left[\frac{1}{2} \|\mathbf{x} - \mathbf{D}\boldsymbol{\alpha}\|_2^2 + \kappa \Omega_{\mathcal{G},\eta}(\boldsymbol{\alpha}) \right] \quad (\kappa > 0). \tag{5.29}$$

The $\boldsymbol{\Omega}$ regularizer has two purposes, in accord with [31]. It enforces sparsity and the topography the columns of D are embedded in. If $\mathbf{y}_G$ denotes the vector

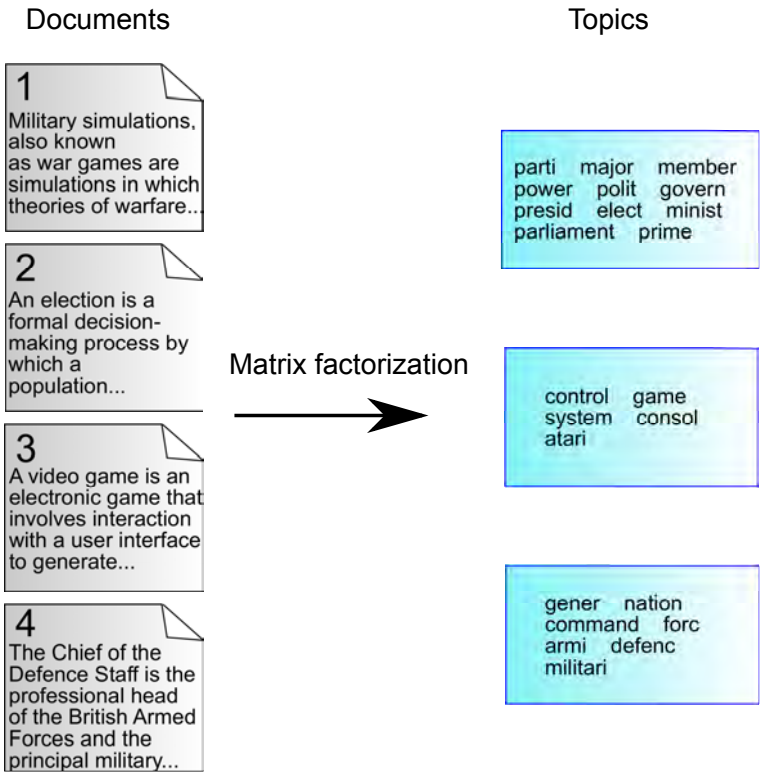


Figure 5.22: **Learning topics from documents** The matrix $\mathbf{X}$ that contains the documents as its columns in a Bag of Words representation is factorized into $\mathbf{X} \approx \mathbf{D} * \mathbf{A}$. As a result, each column of $\mathbf{D}$ will contain a topic. Each column $\boldsymbol{\alpha}^i$ of $\mathbf{A}$ contains the coefficients for topics for the corresponding $\mathbf{x}^i$ document (see also Fig. 5.23). This a hypothetical example.

where all the coordinates that are not in the set $G \subseteq \{1, \ldots, n\}$ are set to zero, then

$$\Omega_{\mathcal{G},\eta}(\mathbf{y}) = \left\| (\|\mathbf{y}_G\|_2)_{G \in \mathcal{G}} \right\|_\eta = \left[\sum_{G \in \mathcal{G}} (\|\mathbf{y}_G\|_2)^\eta \right]^{\frac{1}{\eta}} \quad \eta \in (0, 1] \tag{5.30}$$

If topography is not introduced and thus there are no neighbors, that is, for all $i \in \{1, \ldots, n\}$, $I_i = \{i\}$, and if $\eta = 1$, then the optimization task relaxes to the l_1-norm based Lasso optimization task. If neighborhoods are included and I_is have more than a single element, then the few element choice generalizes to few group choice. However, elements in each group $\boldsymbol{\alpha}_{\boldsymbol{I_i}}$ are subject to the ℓ_2 norm and the selected groups can have dense representations. This allows us to incorporate prior knowledge about the dictionary elements (every coordinate of

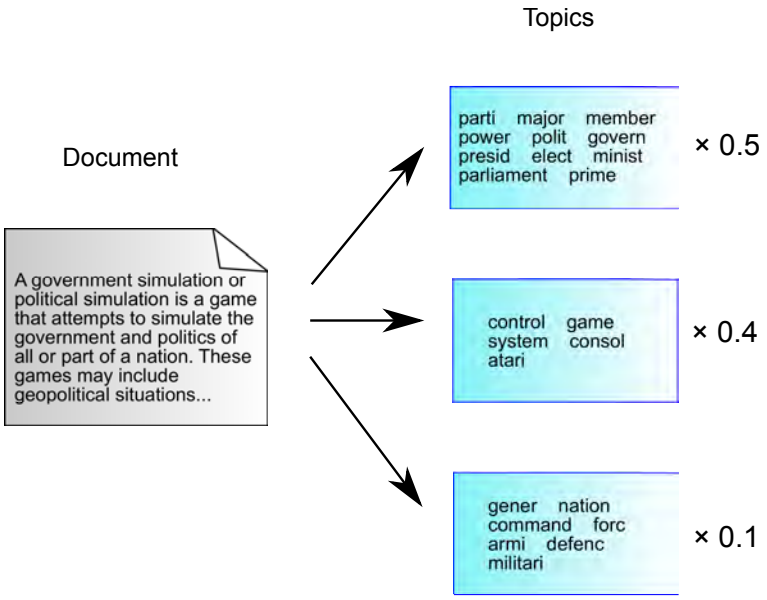


Figure 5.23: **Breaking down a document into topics** A document $\mathbf{x}^i$ is analyzed as a linear combination of topics. $\mathbf{x}^i$ is synthesized as $\mathbf{x}^i \approx \mathbf{D} * \boldsymbol{\alpha}^i$. $\boldsymbol{\alpha}^i$ contains a coefficient for each topic $\mathbf{d}^i$. $\boldsymbol{\alpha}^i_j$ tells us about the extent document i is about topic j. This a hypothetical example.

α corresponds to exactly one column in D).

The other parameter of $\boldsymbol{\Omega}$ is η. If $\eta = 1$, then we sparsify the α_{I_i} in l_1-norm. Decreasing η leads to more aggressive sparsification.

In matrix notation, our task is to solve the following optimization problem:

$$J_{\lambda,\kappa,\eta,\mathcal{G}}(\mathbf{D},\mathbf{L}) = \left[\frac{1}{2}\left\|\mathbf{X}-\mathbf{DL}\right\|_F^2 + \kappa\Omega_{\mathcal{G},\eta}(\mathbf{L})\right] \to \min_{\mathbf{D}\in\mathcal{C},\mathbf{L}\in\mathbb{R}^{k\times n}}, \tag{5.31}$$

We imposed additional constraints on $\mathbf{D}$ and $\mathbf{A}$. For every element d_{ij} of $\mathbf{D}$, $d_{ij} \geq 0$. Also, for every element α_{ij} of $\mathbf{A}$, $\alpha_{ij} \geq 0$. We represent topics in a mixture of topics model: every word can only contribute positively to a topic, and every topic can only contribute positively to a document. So the problem we are solving is a nonnegative matrix factorization problem. There is also a normalization constraint on $\mathbf{D}$: for every $\mathbf{d}^i$ $(i \in \{1,\ldots,n\})$, $\sum_{j=1}^{m} d_{ij} = 1$.

It is a relevant development in our other project that our iterative learning algorithm is online: it can process the $\mathbf{x}^i$-s one-by-one. This is a tremendous advantage compared to batch algorithms: our memory footprint is decreased tremendously, as we do not have to keep the whole $\mathbf{X}$ matrix in memory.

The algorithm learns $\mathbf{D}$ and $\mathbf{A}$ simultaneously. It is very similar to Algorithm

1 in [43], with the following difference. In one iteration, there are two main steps:

1. We optimize the $\boldsymbol{\alpha}_t$ hidden representation for the current $\mathbf{x}_t$ input using the $\mathbf{D}_t - 1$ obtained in the previous iteration.

$$\boldsymbol{\alpha}_t = arg \min_{\boldsymbol{\alpha} \in \mathbb{R}^k} \left[\frac{1}{2} \|\mathbf{x}_t - \mathbf{D}_{t-1}\boldsymbol{\alpha}\|_2^2 + \kappa \Omega_{\mathcal{G},\eta}(\boldsymbol{\alpha}) \right] \tag{5.32}$$

2. We update $\mathbf{D}_t - 1$ using the previous $\{\boldsymbol{\alpha_i}\}_{i=1,\ldots,t}$

$$\hat{f}_t(\mathbf{D}) = \frac{1}{t} \sum_{i=1}^{t} \left[\frac{1}{2} \|\mathbf{x}_i - \mathbf{D}\boldsymbol{\alpha}_i\|_2^2 + \kappa \Omega_{\mathcal{G},\eta}(\boldsymbol{\alpha}_i) \right] \rightarrow \min_{\mathbf{D} \in \mathcal{C}} \tag{5.33}$$

The algorithm works with mini-batches: it processes k inputs in an iteration, where k is a parameter.

5.6.2 The modifications to the algorithm

The modifications of the algorithms concern the three constraints. The two constraints imposed on the elements of $\mathbf{D}$ and $\mathbf{A}$ (i.e., every $d_{ij} \geq 0$ and every $\alpha_{ij} \geq 0$) were dropped. So the matrix factorization is no longer a nonnegative matrix factorization. The reason for this change was practical: the speed of the algorithm can be significantly improved, without significant change to the results. Even without these constraints, most of the elements d_{ij} and α_{ij} are positive, so the results do not change.

The second change is more significant. We change the normalization constraint on D: for every $\mathbf{d}^i$ ($i \in \{1, \ldots, n\}$), $||\mathbf{d}^i||_2 = 1$. Instead of using the l_1 norm, we use the l_2 norm. Because of the l_2 norm, the topics can be more dense than with the l_1 norm. The denser topics allow a topography to be formed, because there are much more nonzero coordinates where topics can overlap.

Results with the new algorithm

We conducted experiments on pages uniformly sampled from Wikipedia. We have experimented with the following topography. The topics are placed on a hexagonal grid on a torus. Every $\boldsymbol{\alpha}_i$ has exactly 6 neighbors (Fig. 5.24).

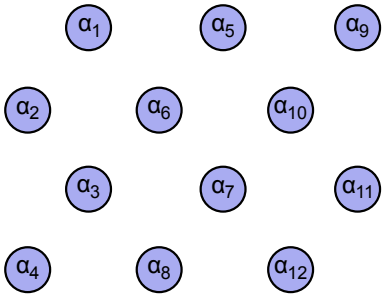


Figure 5.24: **The hexagonal grid the topics are embedded in.** Each coefficient in α represents the contribution of a topic to a document. Each α_i has exactly six neighbors. For example, the neighbors of α_6 are α_1, α_2, α_3, α_7, α_{10}, α_5. The grid is placed in a torus, e.g., α_1 and α_9 are neighbors.

The hexagonal grid is 20-by-20. A 6-by-7 part of this topography is shown on Fig. 5.25. The topics remained well-defined, and now they are embedded in a topography.

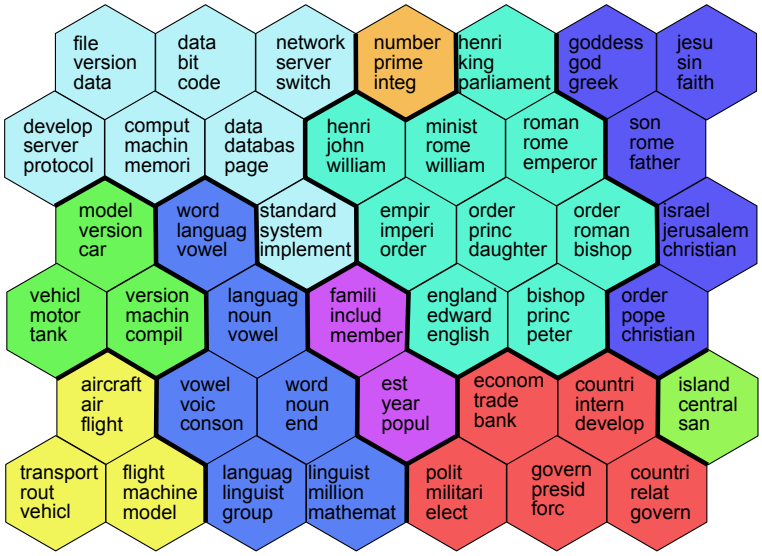


Figure 5.25: **The results of the new, refined algorithm.** The figure shows a part of the topography Online Structured Dictionary Learning generated from a set of documents. Each of the topics are represented by the three words with the largest weights. The colors were assigned to the nodes manually to visualise broad topics. It can be seen that topics that belong to the same broad topic are near each other. Please note that the words are stemmed.

5.7 Interpreting text fragments

In the previous semester, we started analyzing the content of documents by breaking them down into topics. In the first phase of this semester, we improved our algorithm to assign a topography of topics to a set of documents. In the second phase, we have taken this notion one step futher: we introduced topographies of sense-topics that can model the content of single documents, or smaller text fragments.

This new model of documents combines the background knowledge of Wikipedia with the topic topographies we can generate with OSDL to interpret text fragments. The interpretation is a topography of sense-topics. Sense-topics are sense vectors that determine a broader concept or topic. Each word in the text fragment is represented by a combination of these sense-topics embedded in the topography.

The algorithm that generates a topography of sense-topics for a set of words W is an application of Online Structured Dictionary Learning (Sec. 5.6.1) on ESA vectors (i.e., sense vectors that ESA assigns to words). The columns of the matrix X the algorithm factorizes is filled with the vectors ESA assigns to each word $w \in W$. After running the OSDL algorithm on this matrix X, we D will contain a topography whose topics consist of senses, and $\boldsymbol{\alpha}^i$ contains the sense-topic representation of the ith word. (Fig. 5.26)

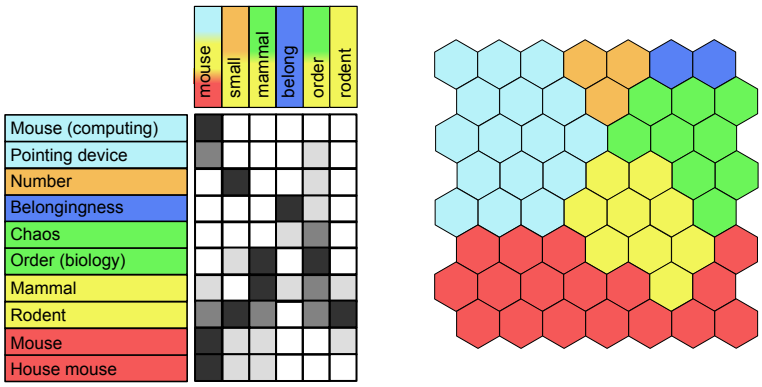


Figure 5.26: **Topography of sense-topics** The columns of the matrix X on the left side contains ESA vectors of words. In this example, the words are from the sentence "A mouse is a small mammal belonging to the order of rodents.". The matrix is factorized using OSDL, $X = DA$. The columns of D contain the sense-topics of the topography on the right side. The columns of A, $\boldsymbol{\alpha}^i$ contain the sense-topic representation of each word. On the figure, the arrows starting from the word "mouse" denote such a representation.

In the next section, we describe the algorithm that generates a sense topography from a collection of words, then we review the two applications we already

explored with good results. The first application decomposes the sense vectors ESA assigns to words according to meaning using the fact that each individual word is represented by a collection of sense-topics. The second application interprets text fragments by finding the most important sense-topics in the text.

5.7.1 Breaking down sense vectors by meaning

Explicit Semantic Analysis assigns sense vectors to words that consist of all the possible meanings of a word. This property of ESA makes it unsuitable to differentiate between different meanings of the same word, and, at the same time, makes it less precise.

In the first application of our topography of sense-topics, we separate the different meanings of sense vectors assigned to words by ESA. We work with a collection of independent words (i.e., the words that are not part of the same text fragment), and put their ESA vectors into the columns of the matrix X, then factorize this matrix into $X = DA$ using OSDL. (Fig. 5.27) The resulting sense-topics mirror the different meanings of the ESA vectors. It is preferable to choose the words such that the different meanings are spread across them.

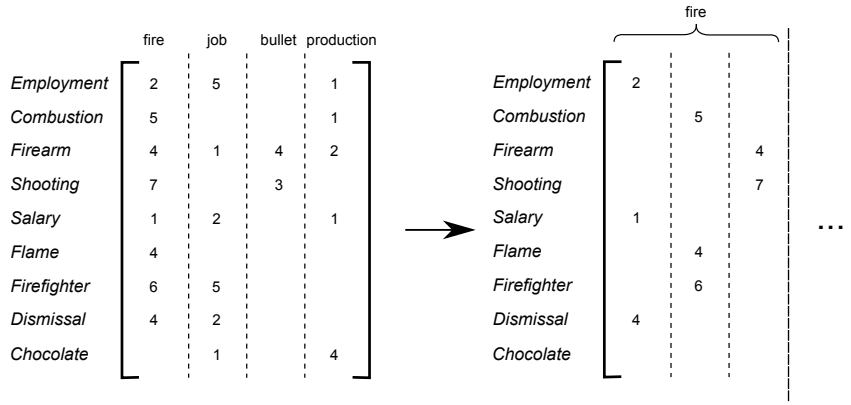


Figure 5.27: **Breaking down sense vectors according to word senses.** The ESA vectors of independent words are put into a matrix X, then this matrix is factorized using OSDL. The resulting sense-topics mirror the different meanings of the ESA vectors. It is preferable to choose the words such that the different meanings are spread across them (e.g., the meanings of fire are contained in the words, "bullet", "job", etc.

The jth coordinate of $\boldsymbol{\alpha}^i$ contains the association strength between the ith word in the matrix and the jth sense topic, and $\boldsymbol{d}^j$ contains the corresponding topic. Let J_i denote the set of nonzero coordinates of $\boldsymbol{\alpha}^i$

$$J_i = \{j | \boldsymbol{\alpha}^i_j \neq 0\} \tag{5.34}$$

As $\boldsymbol{\alpha}^i$ is sparse, J has a small number of elements. There are cases where $\boldsymbol{\alpha}^i$ is not sparse, but it is compressible. In this case we define J_i to contain the indices of the l largest values of $\boldsymbol{\alpha}^i$, where l is the number of sense vectors we want to decompose $\boldsymbol{\alpha}^i$ into. The sense vectors in the decomposition of the ESA vector $\boldsymbol{x}^i$ are the elements of the set

$$\{\boldsymbol{d}^j | j \in J_i\} \tag{5.35}$$

Algorithm 2 (Breaking down sense vectors according to meaning)

1: **Input:** A matrix $X \in \mathbb{R}^{m \times n}$ of ESA vectors, $X = [\boldsymbol{x}^1, \boldsymbol{x}^2, \ldots, \boldsymbol{x}^n]$
2: Factorize X with OSDL: $X = DA$, $D = [\boldsymbol{d}^1, \boldsymbol{d}^2, \ldots, \boldsymbol{d}^k]$ contains the sense-topics in the topography, $A = [\boldsymbol{\alpha}^1, \boldsymbol{\alpha}^2, \ldots, \boldsymbol{\alpha}^n]$, k is the number of sense-topics
3: **for** i = 1, 2, ..., n **do**
4: $J_i = \{j | \boldsymbol{\alpha}^i_j \neq 0\}$
5: $DECOMP_i = \{\boldsymbol{d}^j | j \in J_i\}$
6: **end for**
7: **Output:** The decompositions $DECOMP_i$ for each word.

An example: decomposing the ESA vector of the word "fire"

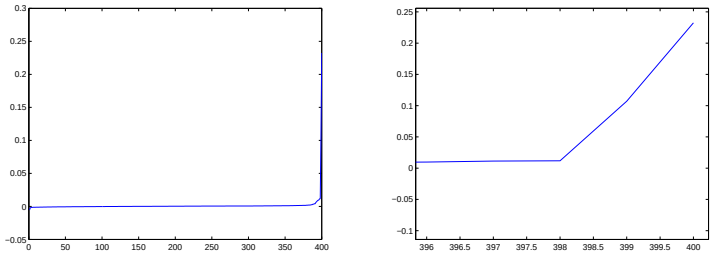
In the following, we demonstrate the algorithm on a concrete example. We decompose the ESA vector of the word "fire" into three meanings:

- the phenomenon of combustion manifested in light, flame, and heat
- the firing of weapons
- to dismiss from a position

Fig. 5.28 shows the coefficients in $\boldsymbol{\alpha}^i$, where i is the column the word fire is in, sorted in ascending order. It can be seen, especially from the figure that only shows the largest coefficients, that there are only three that are not zero. Table 5.7 shows the sense-topics these three coefficients are associated with. They describe the three different meanings of the word "fire".

5.7.2 Assigning sense vectors to text fragments

Explicit Semantic Analysis assigns sense vectors to text fragments by averaging the ESA vectors of all the words in the text. As the ESA vectors contain all the meanings of a word, when we average these vectors, the result will not only represent the meaning of the text, but also alternative meanings based on the other meanings of the words. As the number of words increases, the meaning of the text becomes less and less precise.



(a) the coefficients in $\boldsymbol{\alpha}^i$, (b) the largest coefficients sorted

Figure 5.28: **The coefficients in $\boldsymbol{\alpha}^i$.** The figure shows the coefficients in $\boldsymbol{\alpha}^i$, where i is the column the word fire is in, sorted in ascending order. It can be seen, especially from the figure that only shows the largest coefficients, that there are only three that are not zero.

Flame	Shoot	Dismiss from a position
Chronology of Provisional...	United States Marine...	History of private equity...
List of shipwrecks	Advanced Landing Ground	Private equity in the...
List of accidents and...	Eighth Air Force	Hedge fund
List of illuminated...	United States Air Force	Madoff investment scandal
List of accidents and...	USS America (CV-66)	Subprime mortgage crisis
List of accidents and...	101st Airborne Division	Youth Criminal Justice...
330th Bombardment...	USS Beale (DD-471)	Emergency Economic...
United States Marine...	RAF Alconbury	Enron scandal
List of maritime disasters	Battle of Jutland	Income trust
USS America (CV-66)	Battle off Samar	Financial crisis of...

Table 5.7: **The three sense-topics associated with nonzero coefficients.** They contain concepts that are related to the three different meanings of the word "fire".

The topography of sense-topics allows us to deal with this problem, and assign a more precise sense vector to a text fragment that contains less superflous senses.

We put the ESA vectors of all the non-stopwords in the text into the columns of the matrix X, then factorize this matrix into $X = DA$ using OSDL. We choose the most significant sense-topics as follows. First, we obtain a single vector by summing the coefficients of all the $\boldsymbol{\alpha}^i$

$$\boldsymbol{\alpha} = \sum_{i=1}^{n} \boldsymbol{\alpha}^i \tag{5.36}$$

The vector $\boldsymbol{\alpha}$ shows how significant each of the sense-topics is: $\boldsymbol{\alpha}_j$ is the significance of the jth sense-topic in the text fragment. We obtain the sense vector by choosing the l most significant sense-topics, and averaging them. If we let J to contain the index of the l largest values of $\boldsymbol{\alpha}$, where l is a parameter, then

$$\boldsymbol{s} = \frac{\sum_{i \in J} \boldsymbol{d}^i}{|J|} \tag{5.37}$$

Algorithm 3 (Assigning sense vectors to text fragments)

1: **Input:** A matrix $X \in \mathbb{R}^{m \times n}$ that contains the ESA vectors of the words in the text fragment, $X = [\boldsymbol{x}^1, \boldsymbol{x}^2, \ldots, \boldsymbol{x}^n]$, l, the number of sense-topics we use in the construction of the sense vector

2: Factorize X with OSDL: $X = DA$, $D = [\boldsymbol{d}^1, \boldsymbol{d}^2, \ldots, \boldsymbol{d}^k]$ contains the sense-topics in the topography, $A = [\boldsymbol{\alpha}^1, \boldsymbol{\alpha}^2, \ldots, \boldsymbol{\alpha}^n]$, k is the number of sense-topics

3: $\boldsymbol{\alpha} := \sum_{i=1}^{n} \boldsymbol{\alpha}^i$

4: let J to contain the indices of the l largest values of $\boldsymbol{\alpha}$

5: $\boldsymbol{s} := \frac{\sum_{i \in J} \boldsymbol{d}^i}{|J|}$

6: **Output:** s, the sense vector associated with the text fragment

Results

We have tested the performance of the algorithm an article titled "Hubble Finds Granddaddy of Ancient Galaxies", attached in the Appendix (Sec. 5.7.3).

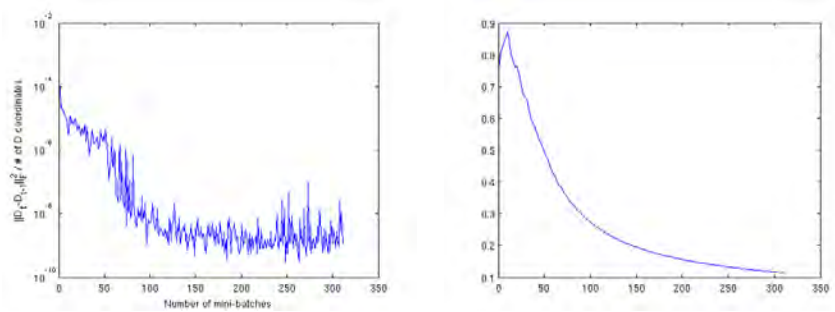
The convergence of the algorithm can be seen on Fig. 5.29. The two figures show the change in D,

$$||D_t - D_{t-1}||_F^2 \tag{5.38}$$

, and the average reconstruction error,

$$\frac{\sum_{i \in processedindices} ||\boldsymbol{x}^i - D\boldsymbol{\alpha}^i||_2}{|processedindices|} \tag{5.39}$$

, as the algorithm progresses. It can be seen that the D changes less and less, and that the reconstruction error decreases as the algorithm progresses.



(a) Change in D (b) Reconstruction error

Figure 5.29: **The algorithm converges.** The two figures show the change in D, and the average reconstruction error, as the algorithm progresses. The number of minibatches is on the horizontal axis. It can be seen that the D changes less and less, and that the reconstruction error decreases as the algorithm progresses.

The topography and the most important sense-topics are illustrated on Fig. 5.7.2, and on Table 5.8. The numbering of the sense-topics are the same on both the table and the figure. The 1st sense topic is the most significant (i.e., it has the largest weight in $\boldsymbol{\alpha}$), and so on. From the figure, it can be seen that the topics are concentrated in clusters, but the topics that describe the article best (the 1st and the 7th topic) are not clustered more closely than the other topics. We are currently in the process of running experiments to achieve this separation, thereby obtain even more accurate sense vectors for the text fragments (Table 5.9).

The sense vectors of ESA and our method are compared in Table 5.10. The senses are numbered according to their significance. Only the 20 most significant senses are shown. It can be seen that in our method, the first five most significant senses are correct, whereas in ESA, only two of them are. Among others, we have successfully filtered out the irrelevant senses about tea.

Topic number	Senses in the topic
1	Milky_Way Galaxy Atlas_of_Peculiar_Galaxies Spiral_galaxy Dark_matter
2	Paper_size Largest_organisms Perceived_visual_angle Depth_of_field File_Allocation_Table
3	Dark_matter The_Dark_Tower_(series) After_Dark_(TV_series) Eisner_Award Dark_Bring
4	Jumanji_(TV_series) Poker_probability_(Omaha) Coach_Trip Academic_term History_of_the_Kansas_City_Chiefs
5	Varese_Sarabande Euclidean_algorithm Classic_Hits_FM Queen_(band) The_Miracles
6	History_of_wood_carving American_Beauty_(film) Orson_Welles Aesthetics Florence
7	Hubble_Space_Telescope History_of_the_telescope Hubble's_law Optical_telescope Lovell_Telescope
8	BKL_singularity Electron Hindu_chronology Babe_Carey Ganymede_(moon)
9	Shift_work Manual_transmission High_German_consonant_shift British_Columbia_Ambulance_Service Ages_of_consent_in_North_America
10	Carburetor Oil_tanker Tide Natural_gas Natural_gas_storage

Table 5.8: **The 10 most significant sense-topics, in order of significance** We only show the first five most significant senses in each topic. The first and seventh topics are the most relevant to our article.

1st sense-topic	7th sense-topic
Milky_Way	Hubble_Space_Telescope
Galaxy	History_of_the_telescope
Atlas_of_Peculiar_Galaxies	Hubble's_law
Spiral_galaxy	Optical_telescope
Dark_matter	Lovell_Telescope
Andromeda_Galaxy	Great_Observatories_program
Chronology_of_Star_Wars	Astronomy_in_medieval_Islam
Hubble_sequence	Telescopic_sight
Galaxy_formation_and_evolution	Hubble_Deep_Field
Rare_Earth_hypothesis	Hubble_sequence
Non-standard_cosmology	Amateur_astronomy
Space_science	Redshift
Globular_cluster	Eyepiece
Black_hole	Observational_astronomy
Gart_Westerhout	History_of_Mars_observation
Places_in_The_Hitchhiker's_Guide_to_the_Galaxy	Galaxy
Astronomy	Mills_Observatory
Big_Bang	Astronomical_seeing
Future_of_an_expanding_universe	Nicholas_Mayall
Ancient_(Stargate)	History_of_astronomy

Table 5.9: **The two most relevant sense-topics.**
The table shows the two sense-topics that are most relevant to the article.

	Standard ESA	Our method
1	Tea	Milky_Way
2	Hubble_Space_Telescope	Hubble_Space_Telescope
3	Galaxy	Chronology_of_Star_Wars
4	Tea_culture	Galaxy
5	Licensed_and_localized_editions_of_Monopoly	Dark_matter
6	Glossary_of_cue_sports_terms	Jumanji_(TV_series)
7	California_locations_by_per_capita_income	Escapement
8	Milky_Way	Gart_Westerhout
9	Helium	Hot_dog_variations
10	Dark_matter	Atlas_of_Peculiar_Galaxies
11	Globular_cluster	Non-standard_cosmology
12	Timeline_of_United_States_inventions	Star
13	Santa_Cruz,_California	Extrasolar_planet
14	Pu-erh_tea	Hubble's_law
15	Star	Iggy_Pop
16	Atlas_of_Peculiar_Galaxies	Galaxy_formation_and_evolution
17	Big_Bang	Black_hole
18	History_of_the_telescope	Timeline_of_United_States_discoveries
19	Visual_acuity	Donna_Summer
20	NORAD_Tracks_Santa	Globular_cluster

Table 5.10: **The sense vectors assigned to the article by ESA and our method.**
The senses are numbered according to their significance. Only the 20 most significant senses are shown. It can be seen that in our method, the first five most significant senses are correct, whereas in ESA, only two of them are. Among others, we have successfully filtered out the irrelevant senses about tea.

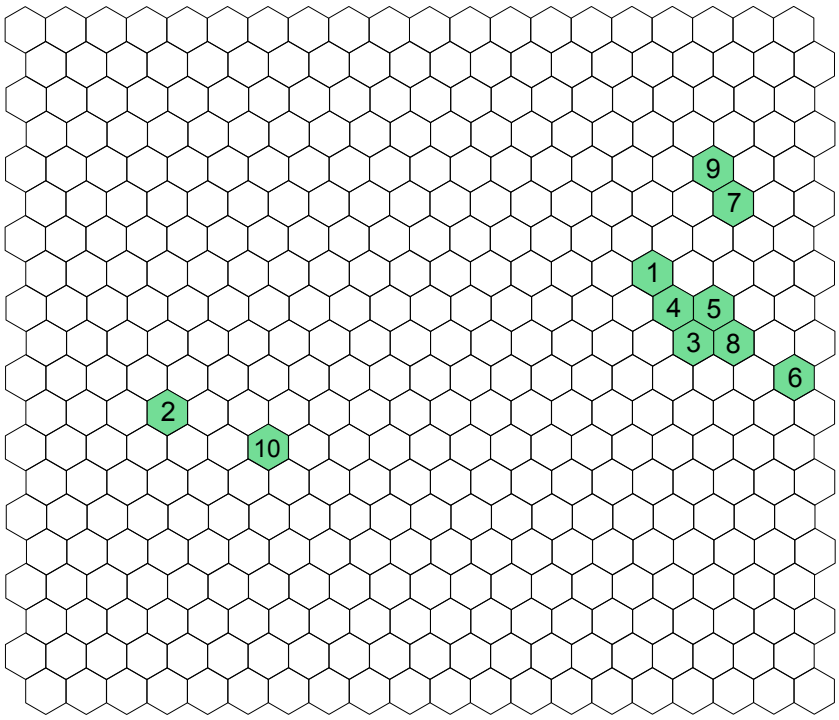


Figure 5.30: **The position of the ten most significant sense-topics on the topography.** The topography is a 20-by-20 hexagonal grid on a torus. The ten most significant topics are colored green, and they are numbered in the order of significance. These topics are detailed in Table 5.8. It can be seen that the topics are concentrated in clusters, but the topics that describe the article best (the 1st and the 7th topic) are not clustered more closely than the other topics.

5.7.3 The article we analyze in Sec. 5.7.2 – Hubble Finds Granddaddy of Ancient Galaxies

Somewhere out in the void - about 13.2 billion light-years away, give or take - is a magnificent red blob that was recently discovered by the Hubble Space Telescope. It's a galaxy - or at least it was; it has long since flashed out of existence - but it is far less beautiful or dramatic than nearly any galaxy the Hubble has spotted before. Its magnificence, instead, comes from its age.

The newly discovered star cluster - a hundred times smaller than our Milky Way - was formed just 480 million years after the 13.7 billion-year-old universe was born, making it the oldest galaxy ever found. As such, it provides astronomers with a first-time glimpse at the universe in its R&D phase, when small, sloppy galaxies were being formed out of hot gas, only to vanish shortly

afterward - leaving the skies free for the huge and mature galactic swirls that would come along later. (See "The Hubble Space Telescope's Greatest Hits.")

Reported in this week's edition of Nature, the galaxy - known, unpoetically, as UDFj-39546284 - had long escaped the Hubble's gaze, and that's no wonder. Even at its best, the 20-year-old telescope never had the acuity to peer so far into space, where the rapid expansion of the universe causes light waves to shift to a deep red. It was only after Hubble's May 2009 upgrade that its Ultra Deep-Field Infrared imager went online. Ultra-deep infrared is exactly what was needed to spot something like UDFj-39546284, but even then, it took about 100 hours of observing time spread across the summers of 2009 and 2010 for the galaxy to be fully visually resolved.

When it was, the findings revealed a lot. The galaxy - or mini-galaxy, as NASA is calling it - is thought to have been just 100 million to 200 million years old when its light began the 13.2 billion light-year journey to Hubble's lens. Its size, shape and the era in which it formed all suggest that it began its life as a mass of gas trapped in a pocket of dark matter - a little like a slosh of tea pooling in the depression of a saucer. (See pictures of five different space programs.)

"We're peering into an era where big changes are afoot," says astronomer and astrophysicist Garth Illingworth of the University of California, Santa Cruz, a co-author of the paper.

The changes were big indeed, but they unfolded slowly. In those early days, stars took about 10 times as long to form as they did in later epochs. When they did form, they were typically part of the blue star class - huge, extremely hot stars, heavy on helium, oxygen and nitrogen. Blue stars are fuel gluttons, lasting only a few million years before ending their lives in massive explosions. (See a brief history of the Hubble Space Telescope.)

It would not be long before stabler, faster-forming stars began popping into being in much larger galaxies as the universe rapidly cooled. Between 480 million and 700 million years after the Big Bang - when UDFj-39546284 was still in the skies - star formation accelerated tenfold. It was then when spiral galaxies and the other glorious formations that define the modern universe appeared.

Just what forces drove those changes are not certain. Hubble has a lot more stargazing to do before more answers are revealed - and a lot more already-gathered images of thousands of other galaxies to analyze. Better still will be the information that comes from the long-awaited James Webb Space Telescope, the Hubble follow-on, which is slated for launch in 2015. (See the top 10 scientific discoveries of 2010.)

"If we go a little bit further back in time, we're going to see even more dramatic changes," promises Illingworth, "closer to when the first galaxies were just starting to form." Not far beyond that lies the dawn of the cosmos themselves.

Chapter 6

Progress in interpreting words

6.1 Translating words to Wikipedia senses

6.1.1 Encyclopedic knowledge - Wikipedia

Wikipedia [62] was launched in 2001 by J. Wales and L. Sanger. Today, Wikipedia is one of the largest encyclopedia of the world. It has 3 million articles in English written in a collaborative manner. Volunteers around the world edit the content of Wikipedia. Although precision of Wikipedia is debated by some and the coverage may not be properly balanced, it is a huge representation of human knowledge and associations. One recent study [24] even found Wikipedia's accuracy to rival that of Encyclopedia Britannica.

One can consider a page in different ways. Clearly, every page is about a particular subject, for example, about the Black-Scholes equation of finance. At the same time, many of the pages are about particular word senses, e.g., there is a page about a bar in the establishment sense, there is one about bar as the counter, etc.

Somehow, these two ends, namely the description of a mathematical construct and the description of word senses everyone knows have many things in common on the Wiki pages. Almost all page contains a summary, the history of the word / phrase / expression, and a detailed description of the concept behind the word / phrase / expression. From the point of view of single words that may have different meanings, Wikipedia pages distinguish the 'senses'. This sense description is much richer (but less precise) than that of WordNet, a lexical database for the English language (`http://wordnet.princeton.edu/`). We can thus identify a concept or word sense with a page in Wikipedia. From now on, a Wikipedia page will be called concept or sense, interchangeably.

Another aspect of Wikipedia is its potential in dialogue systems. Except for the Wikipedia pages themselves, it is rare that a text fragment is strictly about a sense. It is however possible to map a text fragment to Wikipedia senses and to characterize the text by some weights of the Wikipedia senses. Then, having these weighted senses, one can search the database for texts having similar

weighted senses and can use that information for asking or answering questions. This type of machine conversation resembles the Chinese room example (`http://plato.stanford.edu/entries/chinese-room/`) of Searle, meant to argue against the possibility of true artificial intelligence. In essence, we incorporate Wikipedia into our conversation engine to validate the topic of conversation. Details on the utilization of Wikipedia are described below.

The SenseGraph representation

One of the main obstacles of natural language processing is that the same concepts can be described in entirely different words. In the previous semesters, we experimented with various graph representations of texts. In these graphs the vertices represent the words of the texts, and the edges may represent grammatical, lexical, or semantic connections between them.

However, representing texts with words has some drawbacks, because:

- Multiple words can have the same thing (in case they are *synonyms*).
- A single word can mean multiple things (called *polysemy*).

In order to overcome the 'translation' problem amongst different domains, we altered these representations: the vertices of the graphs were changed. Instead of representing *words*, we changed them to *senses*. This transformation – if done properly – has the following advantages:

- It eliminates the synonymity-ambiguity problem, and thus – in principle – it may improve document similarity measures. One expects to recognize that two texts are similar, even if they use different words to describe the same thing; and vice versa, one can recognize differences between texts, even if they use the same words, but in different meanings.
- It makes easier to provide some background knowledge about the things that are mentioned in the text. Note that this task seems unavoidable if one is up to producing a human computer dialogue system.

There are a number of different ways to represent one particular *meaning* of a word (i.e., a *sense*). Perhaps the most popular one is using *WordNet*. WordNet is a lexical database for the English language, which groups words into *synsets*, and provides short definitions for them. One word can belong to multiple synsets. However, using WordNet in our system would have several shortcomings:

- The synsets of WordNet are very fine-grained. For example, the word *source* belongs to no less than nine different synsets as a noun. The differences in the meanings of these senses are so subtle, that occasionally it is hard for ordinary people to classify a particular word of a sentence according to these synsets. Efforts have been made to lower the fine-grained nature of WordNet [56].

- There are a only small amount of sense-tagged texts available for WordNet. *SemCor* (`http://acl.ldc.upenn.edu/H/H93/H93-1061.pdf`, `http://www.cse.unt.edu/~rada/downloads.html`) is such a sense-tagged database, and it is highly precise, but it is too small for machine learning.

- The definitions of the synsets are short. They can be adequate for a human to be able to classify words using them, but they are hardly enough for inference by some kind of artificial intelligence.

Nonetheless, efforts, like the mentioned work on merging word senses [56] and [52], that evaluate a number of methods (`http://wn-similarity.sourceforge.net/`) for measuring the relatedness of concepts in WordNet may eventually succeed to incorporate top level human knowledge into machine evaluations.

WikiSenses

We took a different approach and used the *English Wikipedia* as our source for sense-tagging. In the last few years many papers have appeared that applied it to a host of different problems [44].

Each Wikipedia article defines and describes a single concept or word sense using a hypertext document. So we can disambiguate words in documents to Wikipedia articles denoted by *WikiSenses* from now on.

The first sentence of the article defines the concept in question, and the additional sentences in the same section often provide a more detailed description of it. A Wikipedia article links to other articles. These links link words (the anchor texts) *tagged* to their WikiSenses (the articles). As an example, consider the following sentence:

> Isaac Asimov was a humanist and a rationalist.

The word *humanist* can be a link which points to the sense HUMANISM_(LIFE_STANCE), and the word *rationalist* can be a link pointing to RATIONALIST_MOVEMENT.

There are also redirect pages that link a concept to an article (which is also a concept) . The set of concepts that are mapped to the same concept mean the same. For example, there is a redirect from 'USA' to 'United States'.

Wikipedia also has disambiguation pages `http://en.wikipedia.org/wiki/Disambiguation_page#Disambiguation_pages`. We used these pages as ordinary pages of Wikipedia.

Wikipedia-based word sense disambiguation

We used the *Semantically Annotated Snapshot of the English Wikipedia* [3], because of it is preprocessed in various ways, and so easy to use. We generated data for machine learning from the *hyperlinks* of Wikipedia using the method

described by Mihalcea [45]. We now shortly describe the method, and detail the technical implementation.

We consider Wikipedia articles to be *senses* and hyperlinks to be *learning examples* on which a classifier can learn to disambiguate words. Only those links were processed which suited some requirements. Particularly, we created a list of *permitted words*. A permitted word appears at least once in WordNet *and* at least three times in the *British National Corpus*. This way we limited ourselves to 40,933 words.

The requirements for the links were as follows:

- Their surface forms were required to consist of a single word[1].
- The lemma of this word had to be a *permitted word*.
- The link had to point to an *existing* article, whose length had to have a text of more than *100 words*.

A total of 6,704,196 links met these requirements. We extracted these links, and made feature vectors from them. The features were:

- The article the link was pointing to (following redirects). This is the sense we want to determine.
- The lemma of the surface form of the link.
- The parts-of-speech of the link and the surrounding words.
- A set of words which represents the context of the link. We put the lemmas of the following words into the set:
  - The words that surround the link, three to the left and three to the right (leaving out stopwords).
  - The noun and verb before and after the link (leaving out stopwords).
  - The five most frequent non-stopwords in the whole section that contains the link. We only included words that appear at least three times in the section.

Drawbacks

While the method we used for Word Sense Disambiguation proved to be simple, fast, and fairly precise, it also has some drawbacks, which are difficult to overcome:

- Wikipedia is a very large, manually annotated encyclopedia, but it fails to represent the whole English language. A classifier trained on Wikipedia may fail on texts that are far from being encyclopedic.

[1] We intend to extend our approach to include multi-word phrases.

- The links in Wikipedia tend to point to the correct article, but they also to represent every sense equally. Sometimes, they are biased towards rare senses. For example, the lemma *grip* has multiple meanings, such as the common HANDLE_(GRIP) and the much rarely used GRIP_(JOB) (which means a lighting technician in film industries). In Wikipedia, when this word is used in the sense *handle*, it is rarely linked, because this meaning is trivial; but, when used as *lighting technician*, it is almost always linked. Therefore, in this case training examples are severely biased, and the classifier tends to use the less common sense too frequently.

- The linked words in Wikipedia are mostly *nouns*; linked *adjectives* and *verbs* are rare. Therefore, this method suits the disambiguating nouns.

- The links with lemma *list* point to more than 2500 distinct articles. Almost all of them are *Wikipedia lists*, e.g., LIST_OF_MUSEUMS or LIST_OF_AIRPORTS_IN_ALBERTA. Therefore, the classifier for the lemma *list* is useless.

These drawbacks could be overcome, e.g., by means of combining WikiSenses with other sense evaluation methods, such as the Explicit Semantic Analysis method.

ESA - Explicit Semantic Analysis

Explicit semantic analysis [21,22] is a method to associate senses (=*concepts* in the terminology of the authors) to words. ESA assigns a concept vector to a word, where each concept is weighted by the strength of its association to the word. This is a very simple and highly successful method. It can be applied to many different corpora where documents describe concepts, including Wikipedia. They used Wikipedia as follows:

1. Each Wikipedia article is represented by a vector, where each component corresponds to a distinct word of its text. The value of the component is the TFIDF value of the word that – in loose terms – counts the number of appearances of the word and divides it with the logarithm of the number of documents in the corpus (in this case, Wikipedia) that contain the word. Let $TF(w_i, d)$ denote the term frequency, the times of word w_i occurs in document (or article) d. Let $DF(w_i)$ denote the number of documents that contain word w_i. The TFIDF weight of the word in the document given the database is

$$TFIDF(w_i, d) = TF(w_i, d) * \log\left(\frac{|D|}{DF(w_i)}\right) \tag{6.1}$$

2. For a word, the concept vector is constructed as follows. The weight of a concept in the vector will be the weight of the word in the TFIDF vector of the article that describes the concept.

3. Insignificant associations are discarded: senses with low weights are removed

Semantic relatedness of words can be measured by comparing their concept vectors using traditional similarity measures (e.g., the cosine distance). Comparing documents is also possible with a slight extension.

Reconstruction networks

A twist to the above method is to consider the senses as sources that 'emit' the words according to their TFIDF values. The question is the best linear combination of senses and thus the best linear combination of the inverted index vectors that *together* reproduce (reconstruct) the TFIDF vector of the document. It is advantageous to develop sparse representation, i.e., to diminish small amplitude sense components from the reconstruction procedure. This method was demonstrated in [20] and may improve the ESA results, because it represents the document with a sense vector that *would have emitted* the same TFIDF vector of the document. The reconstruction network architecture and its relaxation equation are shown in Fig. 6.1

6.1.2 Experiments with Sensegraphs

By processing Wikipedia, we created feature vectors for 20,509 distinct lemmas out of which 15,054 had more than one WikiSenses. We trained a *Naïve Bayes* classifier for each of these 15,054 lemmas. The number of senses per lemmas is shown in Fig. 6.2 in decreasing order, whereas the number of training examples per lemma is depicted in Fig. 6.3 in decreasing order, too.

To measure the precision of the classifiers, a 10-fold cross-validation was performed. The the macro-average of the accuracies of the classifiers was 74.41%. Classifiers with poorer performance had fewer learning examples (i.e., the words appeared less frequently in the texts). The micro-average accuracy on the other hand was 84.98%. Figures 6.4 and 6.5 show the statistics of the accuracies of the individual classifiers. The results clearly indicate that classifiers with more training examples and less senses tend to be more precise.

The number of senses actually used by the *Naïve Bayes* classifiers were surprisingly low. For the Reuters Corpus, the number of senses the classifiers mapped to the lemmas is depicted in Fig. 6.6.

The reason of this is twofold:

- The Reuters Corpus is very small compared to Wikipedia, and the articles it contains are similar to each other (they are all financial news). Only a fraction of the concepts that appear in Wikipedia are mentioned in the Reuters Corpus.

- The majority of the WikiSenses appear only very few times, and the Naïve Bayes classifier weights the senses according to their *a priori* probabilities, i.e., that how many times they appear. For example, the lemma

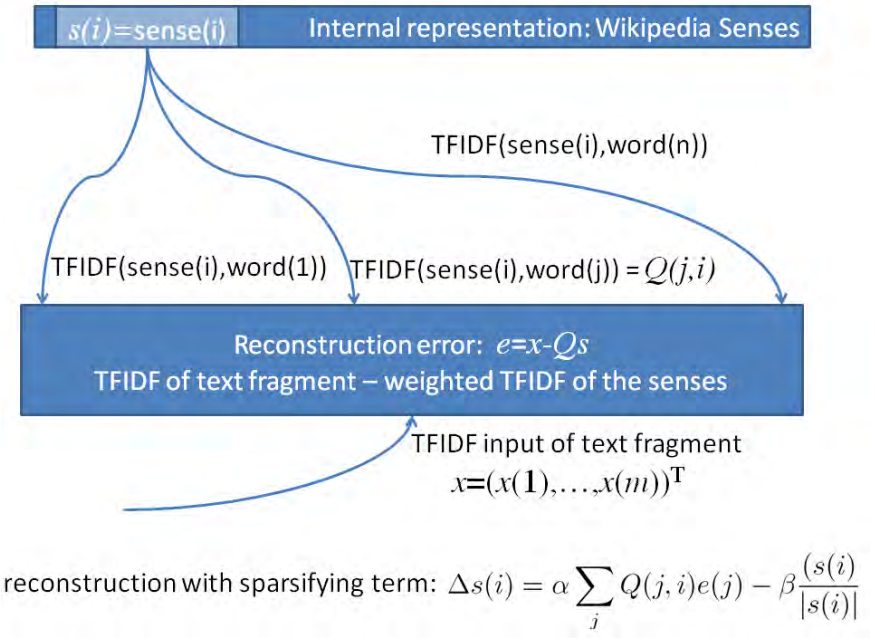


Figure 6.1: **Generative reconstruction network with sparsifying relaxation dynamics.** The network reconstructs its input x by means of the internal representation s. It computes the pseudoinverse of long-term memory matrix Q if $\dim(s) < \dim(x)$, the multiplier of the sparsifying term $\beta = 0$. If $\dim(s) > \dim(x)$ then it sparsifies the representation and computes representation, which cost is (a local) minimum in L_1 norm

abacus has two senses (i.e., in Wikipedia, when the word *abacus* appears, and it is a link, it can point one of two articles): ABACUS and ABACUS_(ARCHITECTURE). We have 69 training examples for the former, but only 7 ones to the latter. Therefore, the Naïve Bayes classifier of lemma *abacus* requires a highly similar feature vector to one of the training examples of ABACUS_(ARCHITECTURE). Otherwise, with high probability, the text will be classified as ABACUS.

6.1.3 Experiments in document categorization

We tested the efficiency of our methods on a document categorization task, a standard benchmark in natural language processing. The goal of this task is to assign the correct category labels to documents from a predefined set of category labels.

We conducted the experiments with two goals: we wanted (i) to determine the utility of different feature representations and (ii) to decide whether semantic

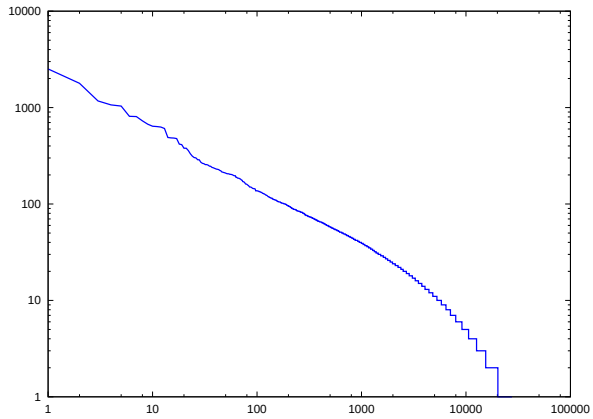


Figure 6.2: **Number of senses for each lemma in decreasing order on log-log scale.** Horizontal axis: number of lemmas ordered according to their senses, vertical axis: the number of the senses of the lemmas. The maximal sense number belonging to a single lemma is 2,500.

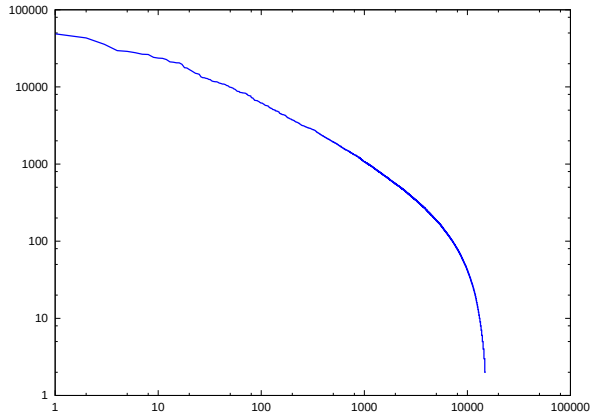
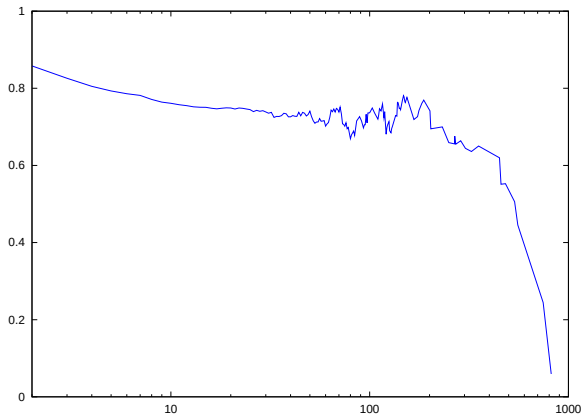


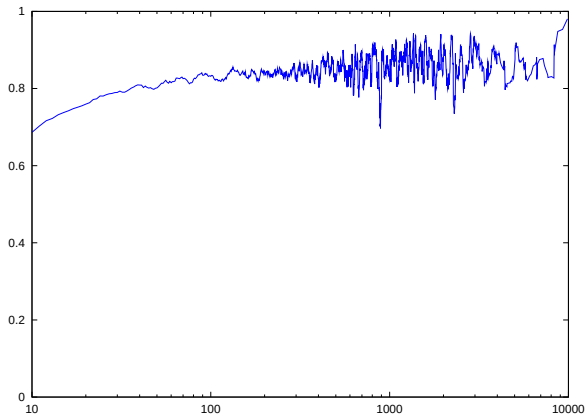
Figure 6.3: **Number of links (i.e., training examples) in decreasing order for each lemma on log-log scale.** Horizontal axis: lemmas sorted by the number of times they appear as a link, vertical axis: the number of the links belonging to the lemma. The maximal number of links belonging to a single lemma is 48,879.

similarity can be inferred by graph kernels using only syntactic information.

We performed document categorization on the Reuters-21578 corpus, and used the 'ModApte' split as described in [32]. We had 9603 training 3299 test documents. To each document there are a number of topics (category labels) assigned.



(a)



(b)

Figure 6.4: **Average accuracies of the classifiers according to 10-fold cross-validation.**
(a): horizontal axis: classifiers ordered according to their number of senses (classifiers having the same number of senses are grouped together). Vertical axis: average accuracy of the groups of classifiers.
(b): horizontal axis: classifiers ordered according to their number of training examples (classifiers having the same number of training examples are grouped together). Vertical axis: average accuracy of the groups of classifiers.

We used the k-nearest neighbor classifier as described in [32]. We used it as a binary classifier; we decide for each topic t and document d whether d belongs to t or not. At the end of the experiment we had a set of topics for each document.

Note that we only wanted to compare feature representations, so we did

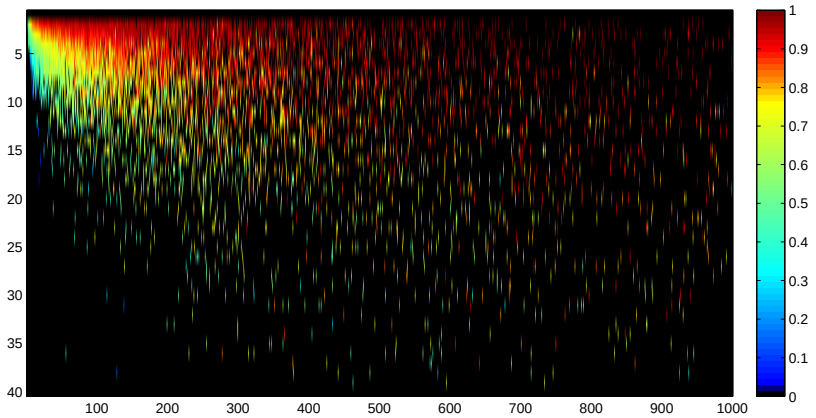


Figure 6.5: **Average accuracies of the classifiers according to 10-fold cross-validation.** We grouped the classifiers which have the same number of senses and training examples. One colored point represents such a classifier group. The vertical coordinate of a point gives the number of senses; the horizontal coordinate gives the number of training examples. The color shows the average accuracy of the classifiers in the group. For example, a yellow point at coordinates Y=5, X=100 means that classifiers which have 5 senses and 100 training examples have an accuracy of 0.75 on average.

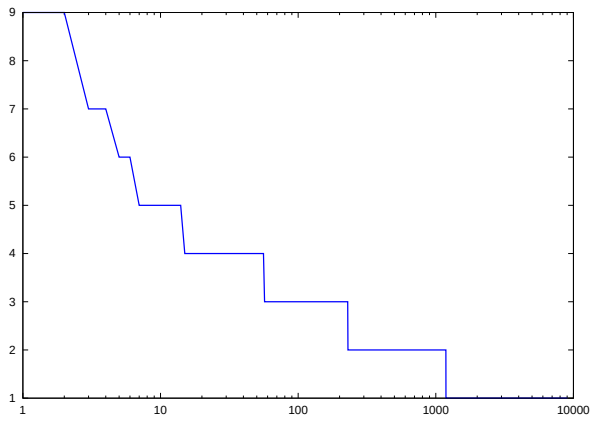


Figure 6.6: **The number of senses mapped to lemmas in the Reuters Corpus on log-log scale.** Horizontal axis: lemmas that (i) appear in the Reuters Corpus, (ii) have at least two different senses sorted by their number of senses assigned to them. Vertical scale: the number of senses.

	kNN - cosine	kNN - dice	kNN - jaccard
Bag of Words	0.758	0.761	0.765
Bag of Sense	0.756	0.762	0.764

Table 6.1: **F-measures of the different methods and feature representations**

not perform sophisticated fine tuning of the parameters. As a consequence, our results are slightly lower than those in the literature, but are not fine tuned for a particular corpus.

We tested the bag of words (BoW) representation, and the SenseGraph representation. For the BoW representation, we preprocessed our documents as described in Sec. 4.2.4 except that we did not extract textual content, since we used text files to begin with. The SenseGraph representation was created as in Sect. 6.1.2. The '*Bag of Senses*' (BoS) representation was created from the SenseGraph after ignoring the graph structure.

We have reached the same performance with the BoS representation as with the BoW one. This means that we could properly replace the words with their senses, a highly desired property for 'translation'. For checking, we also compared the two feature representations using the SVM classifier (Sect 4.2.4) we used in our crawler, and obtained similar results.

A significant result is that using graph kernels with only syntactic information, we were able to achieve an F-measure of 0.654 on both word graphs and SenseGraphs. This indicates that we can infer the topic and semantic relatedness of documents based on their syntactic structure alone.

6.2 Interpreting words as combinations of senses

In the previous semester, we have experimented with word sense disambiguation. We found that many times there is no specific word sense that could identify the meaning of a word. As [1] say, word senses are somewhat arbitrary. In this semester, we have taken a more flexible approach: we do not describe the possible meaning of words as senses fixed by a lexicographer. We describe meaning as a *combination* of senses instead, where each sense can contribute to the meaning of an individual word in the text.

This Section consists of three parts. In the first part, we describe the concepts we use. In the second part, we describe our experiments with extending Explicit Semantic Analysis to describe word meanings as combinations of senses. In the last part, we propose a new architecture based on the observations of the experiments.

6.2.1 Motivation and preliminaries

Motivation

One of the main obstacles of natural language processing is that the same concepts can be described in entirely different words. Representing texts in the traditional manner (i.e., with words) does not represent their meaning properly, because:

- Multiple words can have the same thing (in this case, they are *synonyms*).
- A single word can mean multiple things, thus be ambiguous. (called *polysemy*).

In the previous semester we experimented with the creation of sense representations. This transformation – if done properly – has the following advantages:

- It eliminates the synonymity-ambiguity problem, improving document similarity measures [21]. One expects to recognize that two texts are similar, even if they use different words to describe the same thing. One can also recognize differences between texts if they use the same words, but in different meanings.
- It makes easier to provide background knowledge about the things that are mentioned in the text. This seems necessary to build a human-computer dialogue system.

Supervised methods

A popular way of word sense disambiguation in based on *supervised learning*. For each word, a *classifier* is trained to recognize the different senses of that word based on its *context*. The main problem with this method is the *knowledge acquisition bottleneck*. For each word sense, a large number of training examples must be created manually. In the previous semester, we created a word sense disambiguation system based on [45]. The idea is to consider Wikipedia links as sense-tagged training examples. However, we were not entirely satisfied with the method.

- In this way, we mapped at most one sense to an instance of a word in a text. To give enough background information on a subject, more senses might be needed.
- Moreover, we found that many times there is no specific word sense that could identify the meaning of a word. Describing the meaning of a word as a combination of senses may be more appropriate.
- Only links with one-word anchor texts could be used. The number of training examples was rather low, therefore the precision of the classification was not always satisfactory.

In this semester, we decided to represent senses by vectors of Wikipedia concepts instead of a single Wikipedia concept. We implemented, evaluated and tried to improve a method called Explicit Semantic Analysis to achieve this.

Explicit Semantic Analysis

Explicit Semantic Analysis [21,22] is a method to associate words with senses (=*concepts* in the terminology of the authors). This is a very simple and highly successful method. It can be applied to different corpora, including Wikipedia. We implemented it in the *Java* programming language. We now shortly describe the method.

Basically, we build a semantic interpreter based on the TFIDF values of the words in the corpus. The idea is that if a word appears frequently in a document, then that document (seen as a concept) represents the meaning of the word to some degree. For each word, a sparse vector of concept weights is created. The more frequent the word is in a given document, the larger the corresponding weight will be.

Let m denote the number of distinct words in the corpus, and n denote the number of documents (i.e., the number of concepts). We construct T, an m-by-n matrix. Its rows represent the individual words (denoted by $t_1, \ldots, t_n$), and its columns represent the concepts (whose documents are denoted by $d_1, \ldots, d_n$).

$$T_{i,j} = tf(t_i, d_j) \cdot log\frac{n}{df_i},$$

where *term frequency* is defined as

$$tf(t_i, d_j) = \begin{cases} 1 + \log count(t_i, d_j) & \text{if } count(t_i, d_j) > 0 \\ 0 & \text{otherwise} \end{cases},$$

where

$$df_i = |\{d_k : t_i \in d_k\}|.$$

After this, each column is normalized to disregard differences in document length:

$$T_{i,j} \leftarrow \frac{T_{i,j}}{\sqrt{\sum_{i=1}^{m} T_{i,j}^2}}.$$

This method can be used for measuring document similarity through the senses. Let $w = (w_1, \ldots, w_k)$ denote the words of a (k words long) document. Let $T[w] = (T_1[w], \ldots, T_n[w])$ denote the concept vector of the word w (i.e., the appropriate row of the previously built T matrix). Then, in [21,22] the *semantic interpretation vector* $c = (c_1, \ldots, c_n)^T$ of document d is introduced as

$$c_j = \sum_{i=1}^{k} T_j[w_i] \quad (j = 1, \ldots, n)$$

Similarity measures, such as the cosine function can then be used to establish similarity between documents.

We used a Wikipedia XML dump from February, 2010. We did a simple filtering to discard disambiguation pages and articles that belong to specific dates. Redirects were collected and resolved in a preprocessing step. From the remaining articles we discarded those that were shorter than 300 words or had fewer than 30 incoming and 30 outgoing links, to filter out stubs and other shorter, less significant pages. After that, 200,082 concepts remained. From each remaining article, the stopwords were removed; the remaining words were stemmed with the *Porter* stemming algorithm [54]. There were 1,248,657 different words in Wikipedia; we discarded those who appeared in less than 10 articles. After that, we computed the concept vectors for the 148,928 words that remained.

6.2.2 Experiments

Although ESA has been proven successful in document classification, it has many problems.

- The concept vectors are very fine-grained (e.g., the word 'mouse' has two main senses, but its concept vector contains thousands of concepts). To use ESA for word sense disambiguation, a coarser representation of senses is needed.
- The method does not disambiguate words: it associates the same vector to the word regardless of its actual sense.
- The concept vectors contain many irrelevant concepts (albeit with low weights).

In the following sections we describe how we attempted to overcome these problems.

Concept clustering

Explicit Semantic Analysis associates many Wikipedia concepts to each word. To be able to represent individual senses of the word, the clustering of these concepts seems desirable. For example, in the case of 'mouse', we want the concepts like Pointing device, Drag-and-drop and Computer accessibility to belong to a cluster which represents computer mouse; concepts like House mouse, Mickey Mouse and Rat to belong to a cluster which represents animals.

To cluster the concepts that belong to a given word, the following steps were taken. For each concept, we created a Bag of Words representation of the corresponding article. All words were stemmed with the Porter Stemmer, stopwords were excluded. We created feature vectors from these BOW-representations, with each component of the vector being the frequency of a word. These vectors were normalized using Euclidean norm. A similarity matrix was created:

each entry in the matrix contained the dot product of two BOW vectors. *Spectral clustering* was applied to this matrix.

We also experimented with creating the similarity matrix using the links in the articles in a "Bag of Links"-manner: instead of representing the articles with the words they contained, the articles to which the links in the actual article pointed were considered. The creation of feature vectors from these, and the following steps were the same as in the previous case. As the results were worse than the results with the Bag of Words representation, we do not describe them.

Spectral clustering needs the number of clusters as an explicit parameter. We have made multiple attempts to estimate the correct number of clusters using the multiplicity of the eigenvalue 0 of the Laplacian matrix. However, the similarity matrix was very dense, so the multiplicity was one, showing one big connected component of the neighborhood matrix. We tried to make the matrix sparser, by setting all the entries to zero that were below a *thresholding parameter*. The number of clusters estimated has risen as the thresholding parameter grew, that is, the matrix grew sparser. The optimal value of the thresholding parameter is still an open question. As spectral clustering using different number of clusters provided sensible results of different resolution, we did not pursue this direction further.

We provide the clustering results for some words. We describe the contents of the cluster, then enumerate some examples. We also present diagrams of the similarity matrices which were used during the clustering (Fig. 6.2.2 and Fig. 6.2.2). To depict the structure of the matrices better, we reordered their rows and columns according to the clustering: the documents (or concepts) in the first cluster are represented by the first few rows and columns in the matrix, the documents in the second cluster by the next rows/columns, and so on. The structure of the matrix is block-diagonal. This means that documents (or concepts) in the same cluster are indeed more similar to each other than they are similar to documents in other clusters.

The clustering of the word 'mouse':

- 2 clusters:
  1. *Fiction:*
     Mickey Mouse; Stuart Little (film); Mouse on Mars, Tom and Jerry
  2. *Science and computing:*
     Assistive technology; Mouse (computing); Stochastic matrix, House mouse; Genomic imprinting
- 4 clusters:
  1. *Fiction (especially Disney):*
     Mickey Mouse; Walt Disney; Goofy; Chip 'n Dale
  2. *Fiction (especially Hanna-Barbera):*
     Speedy Gonzales; Tom and Jerry; Spike (Tom and Jerry);

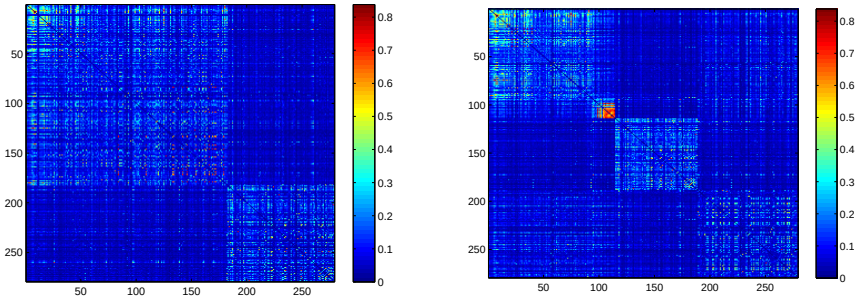


Figure 6.7: **Reordered similarity matrices of 'mouse' for 2 (left) and 4 (right) clusters.** These figures show the similarity matrix of the concepts of the word 'mouse'. The components of the matrix are permuted according to the clusterings. The block-diagonal structure of the matrices can clearly be seen.

List of Tom and Jerry Tales episodes; Stochastic matrix (this is a flaw)

3. *Computing:*
   Assistive technology; Personal computer hardware; Mouse (computing); Repetitive strain injury; Diablo III
4. *Biology, fiction, miscellaneous:*
   Mouse; Maximum life span; Embryonic stem cell; Wild Mouse roller coaster; Stanley Mouse

Another example, the word 'argument':

- 4 clusters:

  1. *Miscellaneous:*
     A Modest Proposal; Chinese room; Direct realism; Redistribution (economics)
  2. *Computing, mathematics:*
     Tail recursion; Lazy evaluation; Function object; Elliptic integral; Fréchet derivative
  3. *Law:*
     Appeal, United States courts of appeals; Compulsory voting; Trial (law); Tax protester arguments
  4. *Logic, paradoxes, apologetics:*
     Cosmological argument; Naturalistic fallacy; Proof by contradiction, Zeno's paradoxes; Reductio ad Hitlerum

The clustering for 'circuit':

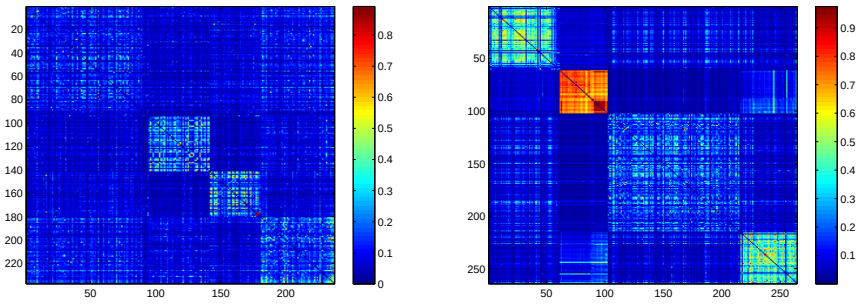


Figure 6.8: **Reordered similarity matrices of 'argument' (left) and 'circuit' (right).** These figures show the similarity matrices of the concepts of the words 'argument' and 'circuit'. The components of the matrix are permuted according to the clusterings. The block-diagonal structure of the matrices can clearly be seen.

- 4 clusters:
  1. *Racing:*
     Le Mans, Open wheel car; Silverstone Circuit; Calder Park Raceway); Hockenheimring
  2. *US counties:*
     Putnam County, Florida; Limestone County, Alabama; Brooke County, West Virginia; Boone County, Indiana; DeKalb County, Indiana
  3. *Electrics:*
     Analog signal; Electrical network; Relaxation oscillator; Dynamic voltage scaling; Circuit rider (religious) (this should go together with the few other Methodist church articles)
  4. *Judicial division, law:*
     United States Court of Appeals for the District of Columbia Circuit; Judiciary Act of 1789; Precedent; Government of Oregon; Methodist Church of Great Britain

Context-sensitive ESA

Explicit semantic analysis associates concept vectors to words without considering the *sense* of the actual instance of the word. For example, the word 'mouse' has two dominant senses: one is the rodent, the other is the computer device. Thus, the concept vector of 'mouse' contains concepts like Pointing device, Drag-and-drop or Computer accessibility as well as House mouse,

MICKEY MOUSE or RAT, each with a high weight; moreover, we assign this same vector to the word regardless its sense. As word senses are correlated with word *contexts* (this is the foundation of most word sense disambiguation algorithms), it would be beneficial to let the context of the word influence its concept vector.

As a first step, we examined the possible benefits of a context-sensitive ESA method by the following word sense disambiguation test:

First, we pick a polysemous target word, e.g., 'mouse' or 'circuit'. We have a corpus for each target word that consists of sentences using the target word in that particular meaning, such as 'The mouse was hungry' or 'He clicked with the mouse'. We would like to create a concept vector representation for each instance of the target word based on its context. We define the context to be a fixed-size window (i.e., the m words before, and the m words after the target word instance). For each instance of the target word, we sum the individual concept vectors of the words in the context (including the target word itself). This sum is a simple context-sensitive concept vector representation. Then we apply a kNN classifier on these vectors see whether it is able to correctly predict the current sense of the word. As a baseline, a simple bag of words-based kNN is used.

Preliminary results were promising. These were based on a small sense-tagged database collected by hand from the Internet by us. For ten ambiguous words, we picked two senses, and collected 15-15 sample sentences for each sense, using *Google search*. This way, we gathered a total of 300 sense-tagged sentences. The results were good, but the small dataset made them a bit noisy. (Figure 6.9 and Figure 6.10)

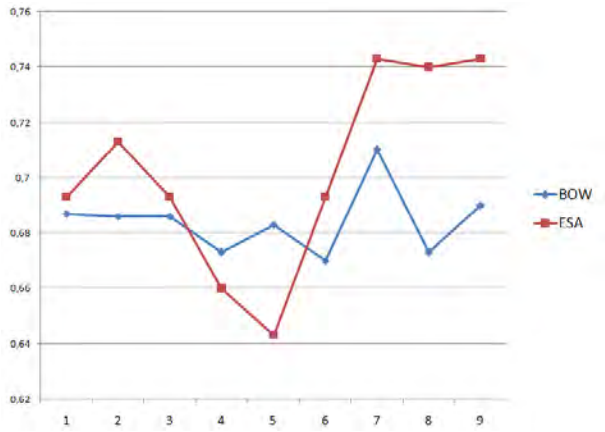


Figure 6.9: **The effect of context radius in word sense disambiguation.** This diagram shows the results of kNN-based word sense disambiguation on the small manually sense-tagged data collected by us. The horizontal axis shows the context radius, the vertical axis shows the accuracy. The ESA-based results show better performance than the BOW-based ones.

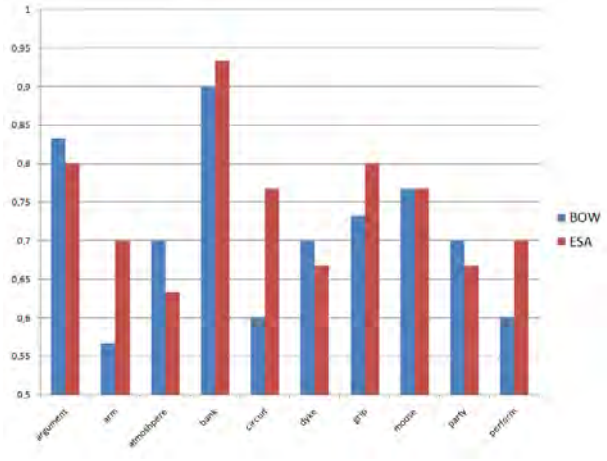


Figure 6.10: **The performance of kNN on the small manually sense-tagged data for context radius of 7.** This diagram shows the results of kNN-based word sense disambiguation on the small manually sense-tagged data collected by us. The horizontal axis shows the individual words, the vertical axis shows the accuracy. The average accuracies were 0.71 (BOW) and 0.743 (ESA).

When performing a larger test on the training dataset of the SENSEVAL-1 English lexical sample task, no improvement over the BOW-based method was observed. (Table 6.2.)

We came to the conclusion that our approach in creating a context-based concept vector representation was flawed, because relying on the TFIDF data of the individual words of the context alone is insufficient (or at least is not better than the much simpler bag-of words model), when disambiguating words. Based on these experiences, we developed a new system, which will overcome these problems. (Section 6.2.4)

6.2.3 Context vector filtering

The concept vector for a word in Explicit Semantic Analysis contains all concepts whose Wikipedia articles mention the given word. For example, the concept vector of 'mouse' contains 3040 concepts, such as HYPERSPACE (SCIENCE FICTION) – albeit with a very low weight – because the article incidentally mentions mouse pointer; the concept of hyperspace in science fiction is, however, not relevant at all in describing the word 'mouse'.

[21] describes a simple heuristic to eliminate spurious associations between articles and words. The method sorts the concepts for the word in descending order of their weights, and then looks for fast drops in the concept scores. The algorithm scans the sequence of concepts with a sliding window of length 100, and truncates it when the difference in scores between the first and last concepts

in the window drops below 5% of the highest-scoring concept for the word.

This method gives somewhat reasonable results, but it only incorporates the TFIDF data. Our method takes into account the structure of the text, and the links in it. We think that a word's concept vector should only contain a concept whose article contains that word as a *keyword*.

We experimented with keyword extraction: we performed some tests with KEA [29], but it needs training data similar to the documents it will be used on to extract keywords well. We came to the idea that in Wikipedia links are only used on important phrases, so an arbitrary article and the anchor texts in it as keyphrases can be used as training examples. We soon realized that this way the usage of KEA can be left out altogether if the corpus we work with is Wikipedia, as there are links in every article. Fortunately, Wikipedia articles are very well structured texts. We consider a word to be a keyword, if it appears in the *lead section* (the first section of the article, before the table of contents), as part of a link's *anchor text*.

The concept vectors filtered this way seem to contain much less irrelevant concepts. A few anomalies, however, were detected. For example, the concept vector of 'house' filtered this way still contains the (not relevant) article SPIDER-MAN'S POWERS AND EQUIPMENT. This happened because the lead section of the article contains the anchor text 'common house spider'. This text was tokenized, and every word-sense pair was used as a training example. Of course, in this case, 'house' has nothing to do with Spider-Man. Considering only the one-word anchor text would discard a considerable percentage of the training examples. Considering 'common house spider' as a whole, we would not be able to decide whether it is a training example for 'common', 'house', or 'mouse'. These experiences led us to design a new architecture. (Section 6.2.4)

6.2.4 The new architecture for senses

Based on the experiences described above, we have designed a new architecture using Wikipedia. The system we came up with incorporates knowledge from the link structure of Wikipedia as well as from the texts of the articles.

First, we create a data structure of many sense-tagged words with contexts. We treat the link structure of Wikipedia as a bipartite graph, where the edges represent the links. The elements of one vertex set are labeled with the anchor texts of the links. The elements of the other vertex set are labeled with the target article of the links. These are the *senses*, or *concepts* in the terminology of Wikipedia-based word sense disambiguation [45]. So one link can be thought of as a sense-tagged example in a corpus. Our goal is to collect as many examples as possible.

To achieve this goal, we build a data structure in two steps. The two steps are marking the links to gather the examples, and gathering the examples.

In the first step, we mark all the links in the bipartite graph that can be used for gathering examples for a given term (the method also works for phrases that consist of multiple terms). First, we take all the links whose anchor text is exactly the term (See Figure 6.11). These are obviously good sense-tagged

examples. After that, we also mark all the links that point to some sense that some already marked link also points to (See Figure 6.12). This way we will mark all the links that have a common sense with some link whose anchor text is exactly the term. These are good examples regardless of their anchor text, because the anchor text is used in the same sense as the term we are collecting examples for. The advantage of this method is that we gather much more correct examples than with previous approaches.

In the second step, we gather the examples from the marked links. We see each link as an example for the given term and the sense it points to. We collect the contexts of the anchor texts of these links, and label them with the article that contained the link (See Figure 6.13).

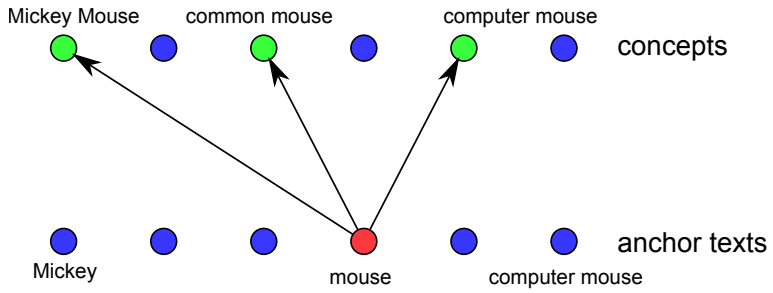


Figure 6.11: **Marking the links for a term-concept data structure.** The dots in the bottom represent the anchor texts of the links. The target articles of the links (i.e., the senses or concepts) are on the top. For example, three links have the anchor text 'mouse', one of which points to the sense MICKEY MOUSE. For the term 'mouse' (marked with red), we mark the links that have the anchor text 'mouse' (concepts marked with green).

The data structure is used to assign senses to terms in a text. We do this by comparing the context of the term to the contexts assigned to the concepts. We will have a weight for each sense, and we can construct a vector, a *sense vector*, that consists of the senses and their weights. If there is one sense whose weight is much larger than the weights of the other senses, then we can discard the vector and keep that one sense as the sense of the term.

The weights of the sense vector can produced with a similarity measure (e.g., the cosine measure), or with a reconstruction network. In the solution with the similarity measure, each context of a sense is compared with the context of the term, and the results are aggregated to yield a weight for the sense. An alternative (and faster) solution is to aggregate the contexts of the sense into one context, and compare only this aggregated context with the context of the term. In the other solution, the reconstruction network is used to find the best linear combination of the senses and thus the best linear combination of the aggregated contexts that reconstruct the context of the term.

The architecture has a number of advantages compared to our previous ap-

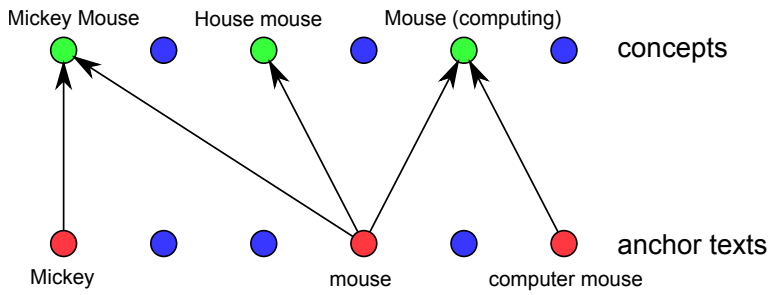


Figure 6.12: **Marking the links for a term-concept data structure.** Continuing the marking the links to gather examples from (see Figure 6.11), we mark all the links that point to some concept some already marked link also points to (the concepts marked with green). For example, we collect the link with anchor text 'Mickey', because it points to a previously marked concept, namely MICKEY MOUSE.

proaches. We expect it to resolve many of these problems we encountered. ESA assigns an overly fine-grained concept representation of a word. Senses that links point to are much more sparse and more coarse. ESA assigns the same vector to a word regardless its actual sense. Our system will perform disambiguation based on the contexts of links in Wikipedia. We used links as training examples before, but the new architecture allows us to gather more. Finally, the concepts we will assign to words will be more relevant than the ones ESA assigns, because the links in Wikipedia denote keyphrases.

There are several ways to extend and improve the architecture:

- The weights of the words that appear in anchor texts, or in the title of the article that contains the link can be enhanced in the BOWs.
- The contexts can be filtered to only contain the words that are characteristic to the term we want to disambiguate. This filtering can be done for example with document frequencies (i.e., only the words which appear in only a few contexts are retained), or with RPCA (Robust Principal Component Analysis).
- The number of learning examples can be improved by collecting all the contexts for frequent monosemous anchor texts. For example, if the anchor text 'computer mouse' links only to a single sense in the entire corpus, MOUSE (COMPUTING), then it can be assumed that the phrase is monosemous (i.e., it is always used in only one sense), and all the contexts can be collected, which contain the phrase 'computer mouse', even if it is not a link (following [39]).
- The words in anchor texts and contexts can be extended with syntactic and POS (part of speech) information.

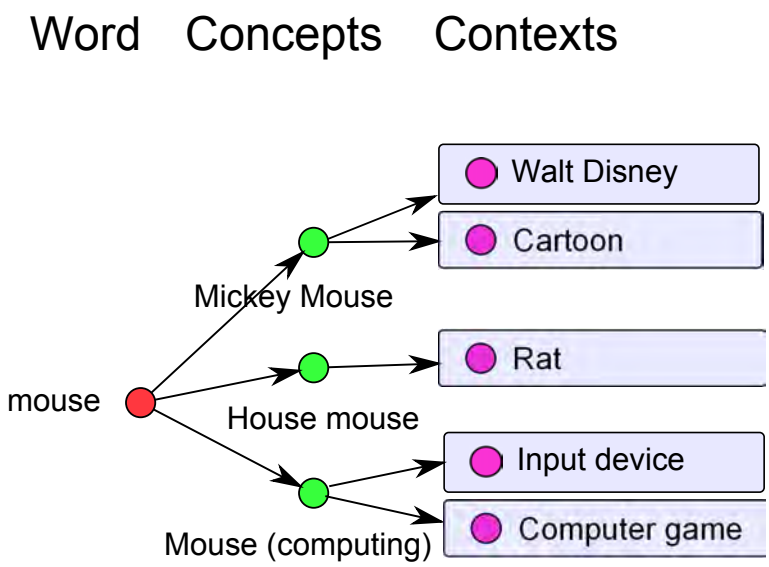


Figure 6.13: **The resulting term-concept data structure.** The term-concept data structure is created from the link structure seen in Figure 6.12. For each term, concepts are assigned, and for each concept, contexts are assigned. For example, the concept MICKEY MOUSE is assigned to the word 'mouse' (because there is a link in Wikipedia which points to MICKEY MOUSE, and whose anchor text contains 'mouse'), and contexts of anchor texts from articles WALT DISNEY and CARTOON are assigned to the concept MICKEY MOUSE (because these links point to MICKEY MOUSE).

- *Second-order features* can be used during the comparison of the contexts: instead of simply comparing the words in the BOW models of the contexts, we can also examine the cooccurence statistics of its words. For example, if the word 'cop' occurs in one BOW, and 'accident' occurs in the other, then they can receive a similarity score if these words frequently appear together with a *third* word (e.g., 'traffic': 'traffic cop' and 'traffic accident' can be frequent collocations).

6.2.5 Additional possibilities

Principal component analysis is a method for *dimension-reduction*: it transforms a number of variables into a smaller number of uncorrelated components. The method is prone to errors, because it uses *least squares* estimation, which can be influenced badly by only a small number of relatively large errors (i.e., outliers). *Robust principal component analysis* [63] is a new method which alleviates this problem: it can recover a low rank matrix from corrupted observations. The errors in the observations can be arbitrarily large, but are assumed to be

sparse. We – and others – applied this method in *computer vision*. The results were highly successful. This promising method can be of great use in computer linguistics as well, but we have not yet managed to apply it in that area because of memory problems. For example, the TFIDF matrices of ESA are very large.

Cyc is the world's largest and most complete general knowledge base and commonsense reasoning engine. We downloaded and examined *ResearchCyc*, which is a version of Cyc, aimed at the research community. The ontology contains an incredible amount of common sense knowledge (like *tree is a kind of plant*, and *plants die eventually*), and the inference engine is able to derive facts using this knowledge (like *trees die eventually*). Some Cyc instances are mapped to Wikipedia concepts and WordNet senses. This kind of general knowledge can be of great use in word sense disambiguation [14], especially in a human-computer dialogue system. For example, if the system is not sure about the current sense of a particular word, it might use Cyc to construct questions to the user which would disambiguate the word ('Is this mouse you are talking about an animal?'). Using Cyc, however, would require a considerable amount of effort, because it is a highly complex system, and many sources report difficulties using it. [11]

6.3 Sparse coding to determine meaning

We define the meaning of a words as a combination of word senses. The problem of assigning this combination, or sense vector, to a word can be cast in a framework that employs sparse representation.

We collect words tagged with senses from Wikipedia using our architecture for interpreting text. One link in Wikipedia can be thought of as an example of a term or phrase (the anchor text) tagged with a sense (the Wikipedia page the link points to). We have a set of words that we want to interpret. For these set of words, we gather all the links in Wikipedia in which they are anchor texts, and store their context (e.g., the k words before and after them) in the columns of a matrix $\mathbf{D}$ in Bag of Words representation.

When we encounter a word w we want to disambiguate, we can ask the following question: What contexts should be combined linearly to get the context of w? (Fig. 6.14) We want to reconstruct the context of w from the linear combination of the least amount of contexts possible. The vector α that contains the coefficients that belong to the contexts in D is the sense vector that belongs to w. If the context of w is x in a Bag of Words representation, then this problem is the same as the one in topic learning, with the exception that here D is given by the contexts gathered from Wikipedia; we do not learn the dictionary (Fig. 6.15). We can solve this problem efficiently.

There are several advantages of this framework. First, it assigns a vector of meanings to a word instead of a single meaning. Second, it assigns meaning across all senses at once. It detects if a word is exchanged with another word based on the context. For example, if in the sentence 'The bomb blew up' we exchange the word *bomb* with e.g., *vase*, the sense vector will still point to the

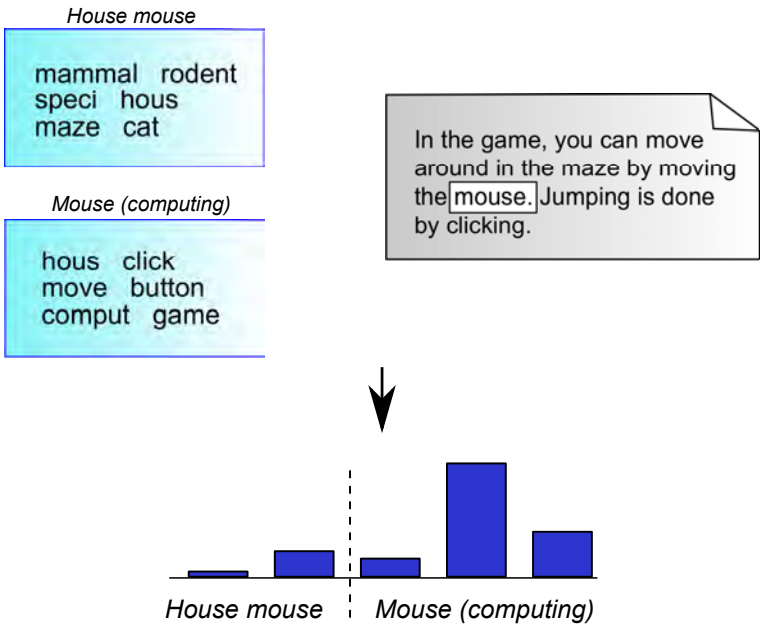


Figure 6.14: **Assigning sense vectors to words.** The algorithm uses sense-annotated contexts we mine from Wikipedia. When we encounter a new word, we can assign a sense vector to it based on these contexts. The activities in the vector represent the extent to which a the corresponding sense contributes to the meaning of the word. See Fig 6.15 for a detailed explanation. This is a hypothetical example.

meaning *bomb* [2]. For the same reason, synonymy and polysemy is resolved. If two words are synonymous, their context and so their vector of meanings will be similar; the surface form of the word does not count. In the same way, if there are two words with the same surface form, but different meaning, their vector of meanings will be completely different.

6.4 Interpreting words

One of the most important questions in natural language processing, and in our project is assigning meaning to words. The previous section shows how to assign meaning to whole text fragments, and how to interpret words as a list of sense-topics. Here we show how to make this interpretation more precise. We assign a sense vector to a word given a list of senses. To solve this problem, we have extended the framework we developed in the previous semester (Sec. 6.3). In the following, we introduce the extended framework, then two

[2] hypothetical example, in real cases more context might be needed

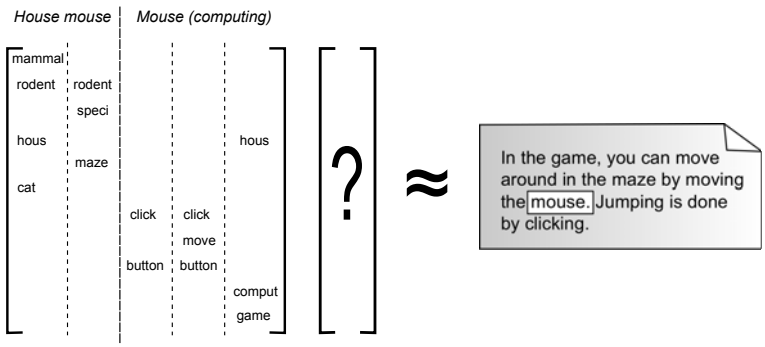


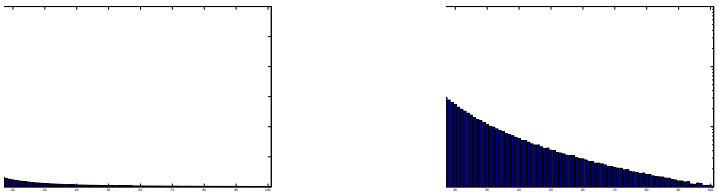
Figure 6.15: **The problem of interpreting text in the context of sparse representations.** We store contexts annotated with senses as the columns of a matrix D. We assign a sense vector to a word w by reconstructing its context as a linear combination of the columns of D. The vector ***alpha*** that contains the coefficients that belong to the contexts in $\mathbf{D}$ is the sense vector that belongs to w. In the example, the word *mouse* belongs to the sense *Mouse (computing)*. Of the five coefficients in ***alpha***, the last three that belong to this sense will be larger than the first two. This is a hypothetical example.

different realizations of the framework, and the results.

6.4.1 A framework to determine the meaning of words

Word sense disambiguation is a well-studied, but still unsolved problem in natural language processing. In WSD, we try to assign a sense to a word that we know from a limited set of senses the word can be used in. The disambiguation of different words are disjunct problems. Two difficulties arise when we try to apply this definition to real world problems. First, the meanings of words are not disjunct. We are depriving ourselves from useful knowledge if we constrain our problem to the surface form of one word. Training examples are generally scarce, even in Wikipedia (Fig. 6.16). The second difficulty is that often the surface form of the word is incorrect. From simple spelling errors to unknown slang and deliberate word exchange, there are many reasons to suspect that using the surface form of the word will lead to incorrect inference.

In the previous semester, we have outlined a framework that defined meaning not as individual word senses, but as a combination of senses. In the sense vector we assign to a word, the combination of its elements (senses) describe the meaning. In addition, the element with the largest coefficient is usually the correct word sense according to the traditional Word Sense Disambiguation problem. This property allows us to compare our algorithms to algorithms used in Word Sense Disambiguation problems. We are using two linear multi-class Support Vector Machines as baselines: one with a one-against-all strategy, and one with a one-against-one strategy.



(a) linear histogram (b) logarithmic histogram

Figure 6.16: **A histogram of the number of training examples per sense in Wikipedia.** Only the articles with more than a hundred non-stopword words were considered. It can be seen (especially from the logarithmic plot) that most of the senses have few training examples.

The first advantage of this framework is that the combination of senses describes meaning more accurately than just a single sense. The second advantage is that we can assign meaning to words we do not know. This is also possible in Word Sense Disambiguation, but as we do not use the word, only its context, the number of possible senses is larger than in WSD. If we assign a combination of senses to a word, we can determine its meaning with less error than if we assign a single sense out of hundreds, maybe thousands of potential candidates.

In every realization of the framework, we start with a matrix $A \in \mathbb{R}^{m \times n}$ whose columns are contexts mined from Wikipedia, labeled both with the word they belong to and their sense. We have a context we want to assign meaning to in $\boldsymbol{y} \in \mathbb{R}^m$. The sense vector consists of k different senses that could be obtained e.g., from sense-topic representation of the word. In the following, we describe the generalized WSD problem, our dataset and feature selection, the methods, and our results.

6.4.2 The generalized WSD problem and the data used in the experiments

We test all the algorithms on the same experiments and the same data. To be able to measure their effectiveness, we use a problem for which a labeled dataset is available.

We have chosen to test the algorithms on a generalized Word Sense Disambiguation problem. In traditional WSD, we determine the sense in which a word is used from a small set of senses. This set is different for each word, so the disambiguation problems of different words are disjunct. There is usually a separate classifier built for each of the words that are to be disambiguated.

In contrast, we are testing on a generalized WSD problem, where the disambiguation of all the words is one problem. In fact, we do not use the surface form of the word at all. We do not build a separate classifier for each separate word surface form; rather, we have one classifier that decides the meaning of all

the words. (Fig. 6.17)

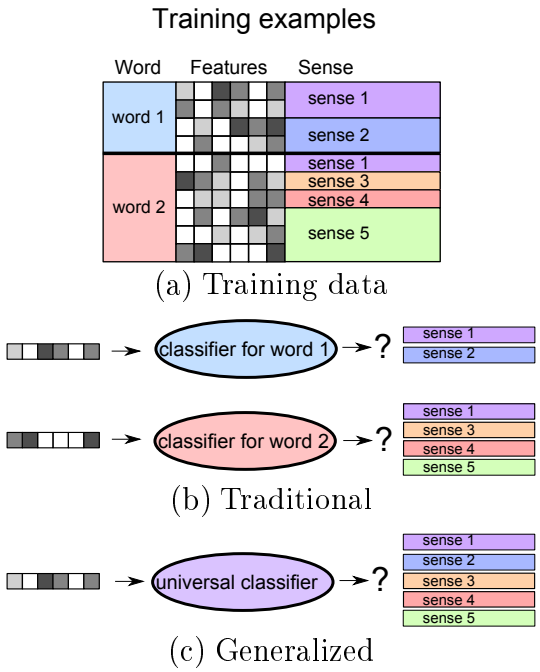


Figure 6.17: **The traditional and the generalized WSD problem** In traditional Word Sense Disambiguation, we have to determine the sense in which a word is used from a small set of senses. This set is different for each word, so the disambiguation problems of different words are disjunct. There is usually a separate classifier built for each of the words that are to be disambiguated. In contrast, we are testing on a generalized WSD problem, the disambiguation of all the words is one problem. In fact, we do not use the surface form of the word at all. We do not build a separate classifier for each separate word surface form; rather, we have one classifier that decides the meaning of all the words.

For this problem, the labeled dataset we use is Wikipedia. We consider the words in the anchor texts semantically annotated. Their sense is the Wikipedia article they link to.

Wikipedia is freely available to download[3] in XML files. We use a version of the English Wikipedia from February 2010. The *SAX parser*[4] is used to handle the XML file. *Apache Lucene*[5] is used to index Wikipedia. Disambiguation pages, and articles that are too small (have less than 200 non-stopwords in their texts) or have less than 20 incoming and 20 outgoing links are not considered

[3] `http://en.wikipedia.org/wiki/Wikipedia:Download`
[4] `http://www.saxproject.org/`
[5] `http://lucene.apache.org/java/docs/index.html`

in further steps. The *Porter Stemming Algorithm*[6] is used for stemming.

To generate the data set, a number of semantically annotated terms are selected randomly that match the following criteria:

- The term has to be a single word that is between 3 and 20 characters long, consisting of the letters of the English alphabet. Hyphens are also allowed.
- To discard very rare words and abbreviations, the word has to occur at least once in *WordNet* and at least three times in the *British National Corpus*[7].
- The word has to be semantically annotated at least 100, at most 200 times.
- The word has to have at least 2, at most 20 different senses (only those senses are considered that the word has at least 2 annotations to).

The links with the resulting terms as anchor texts are extracted from Wikipedia, together with their local contexts of k non-stopwords before and k non-stopwords after the anchor text, where k is a parameter. Stopwords are skipped, so exactly k words are collected in both directions (unless the beggining or the end of the article is reached).

A bag of words representation of the contexts annotated by their sense is placed into the columns of a matrix A. A_{ij} is the number of times the jth word appears in the ith context.

We have mined three datasets from Wikipedia. In each dataset, there are precisely 50 examples for all the senses. In the first and the third dataset, there are 37 senses, and there are 40 in the second (Table 6.3).

6.4.3 Feature selection

Feature selection is desirable for three reasons. First, there are many words among the features that are not real words, but the result of spelling errors, etc. Second, we can improve the accuracy of the algorithms if we can discard the irrelevant features. Third, the time it takes to execute the algorithms decreases as the number of features decreases.

We have examined two methods for feature selection. The first method simply thresholds the words by their frequency. The less frequent words are discarded. The second method is based on the popular TFIDF weighting scheme, but instead of calculating the number of documents the sense appears in, we calculate the number of senses. In the following, we take a look at the data before feature selection, then we describe and analyze the two methods.

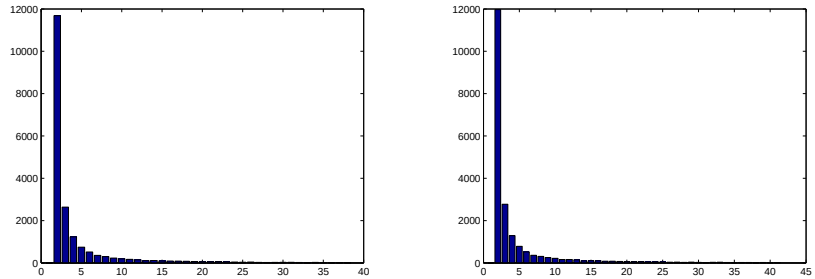
The datasets before feature selection

In this section, we examine the datasets DS1 and DS2 prior to any feature selection. We take a look at two different and important characteristics of these

[6] `http://tartarus.org/~martin/PorterStemmer/`
[7] `http://www.natcorp.ox.ac.uk/`

dataset. Fig. 6.18,shows the distribution of how many senses a word belongs to (i.e., appears in the context of an example that is labelled with the sense). The less senses a word belongs to, the more it helps discriminate between senses. So it is preferable to keep the words that belong to not too many senses.

Fig. 6.19 shows how many words the training examples for each of the senses contain. The senses are on the horizontal axis, and the number of words that appear in all the examples labeled with that sense are on the vertical axis. Preferably, the number of words is not low for any sense. If, for example, there would be no words in the columns of a sense, it would mean that its examples are all zero, so we could tell nothing about it.



(a) DS1 (b) DS2

Figure 6.18: **The datasets before feature selection** The histogram shows the distribution of how many senses a word belongs to (i.e., appears in the context of an example that is labelled with the sense). For example, there are approximately 12000 words that belong to only one sense in both datasets. The less senses a word belongs to, the more it helps discriminate between senses.

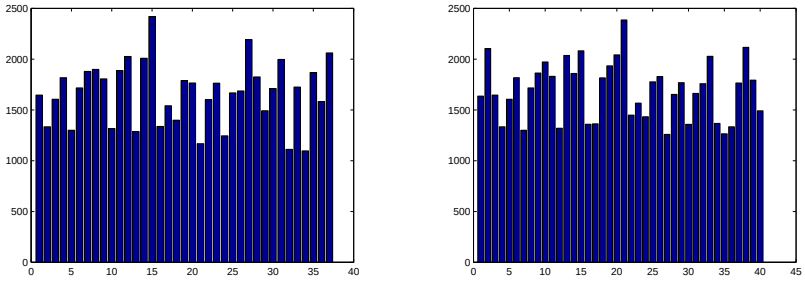
Thresholding by term frequency

The first feature selection technique we review is also the simplest. We simply take the sum of each row in the matrix A, and keep only the words whose corresponding row sum is greater than a predetermined threshold. So we threshold by the frequency of words: we discard every word whose frequency is below the threshold. Fig. 6.20 shows the number of words that remain in the dataset as the threshold grows from 0 to 100.

Fig. 6.21 shows the percentage of nonzero elements in the matrix A as the threshold grows. The sparsity of the matrix can be reduced by thresholding.

Fig. 6.22 shows the distribution of how many senses a word belongs to (i.e., appears in the context of an example that is labelled with the sense). As we increase the threshold, less and less words remain that could discriminate between senses.

Fig. 6.23 shows how many words the examples for each of the senses contain.



(a) DS1 (b) DS2

Figure 6.19: **The datasets before feature selection** The figure shows how many words the examples for each of the senses contain. The senses are on the horizontal axis, and the number of words that appear in all the examples labeled with that sense are on the vertical axis. Preferably, there number of words is not low for any sense.

The number of words (features) do not decrease significantly for any of the senses as the threshold grows.

In conclusion, we can see that there is no difference between how the two datasets respond to thresholding. Thresholding by term frequency allows us to reduce the sparsity of the matrix, but it does not keep the most important words for classification.

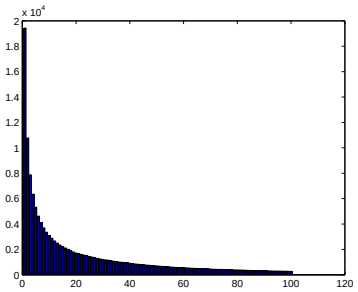
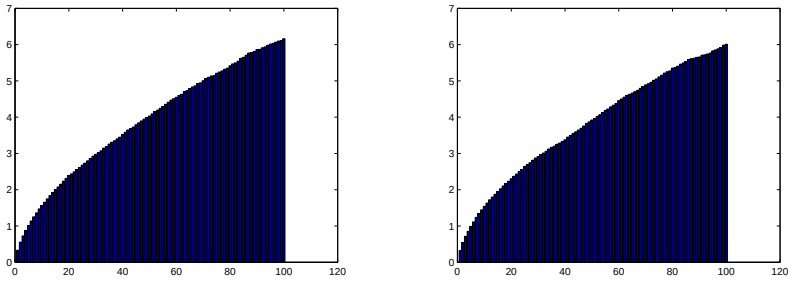


Figure 6.20: **Thresholding by term frequency** The figure shows the number of words that remain in the dataset as the threshold grows from 0 to 100. The threshold is on the horizontal axis, and the number of words is on the vertical axis.



(a) DS1 (b) DS2

Figure 6.21: **Thresholding by term frequency** The figure shows the percentage of nonzero elements in the matrix A as the threshold grows. The threshold is on the horizontal axis, and the percentage of nonzero elements is on the vertical axis. It can be seen that the sparsity of the matrix can be reduced by thresholding.

Term Frequency Inverse Sense Frequency

The second feature selection technique is also based on thresholding, but we generate a new A matrix that contains Term Frequency Inverse Sense Frequency (TFISF) values instead of Term Frequency values. The input of the algorithms is also this new matrix.

TFISF is based on TFIDF. The motivation of TFIDF is that the less documents a word appears in, the more it can be used to distinguish that document. So besides the frequency of the word in the document, the inverse of its document frequency is also important.

For senses, the motivation is that the less senses a word belongs to (i.e., appears in the context of an example that is labelled with the sense), the more discriminative it is for those senses. So we compute the TFISF score as follows.

$$TFISF_{ij} = A_{ij} * \log_2 \frac{k}{s_i} \tag{6.2}$$

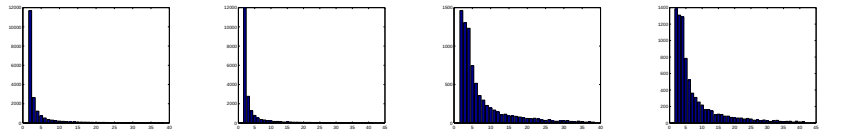
where k is the total number of senses, and s_i is the number of senses whose training examples the ith word appears in.

As the results were practically identical for DS1 and DS2, from here on we only analyze DS1.

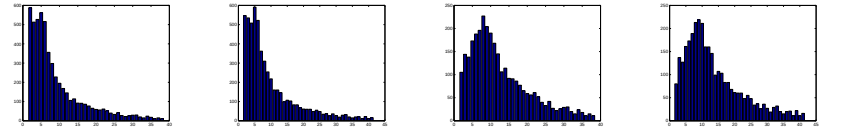
Fig. 6.24 shows the number of words that remain in the dataset as the threshold grows from 0 to 100.

Fig. 6.25 shows the percentage of nonzero elements in the matrix A as the threshold grows. The sparsity of the matrix can not be reduced as well as by thresholding by term frequency.

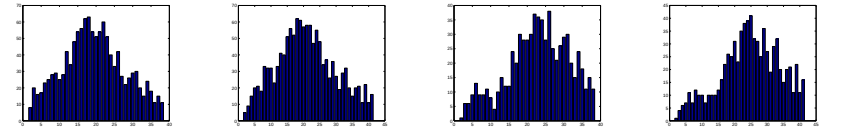
Fig. 6.26 shows the distribution of how many senses a word belongs to (i.e., appears in the context of an example that is labelled with the sense). As we



(a) $t = 1$, DS1 (b) $t = 1$, DS2 (c) $t = 3$, DS1 (d) $t = 3$, DS2



(e) $t = 5$, DS1 (f) $t = 5$, DS2 (g) $t = 10$, DS1 (h) $t = 10$, DS2



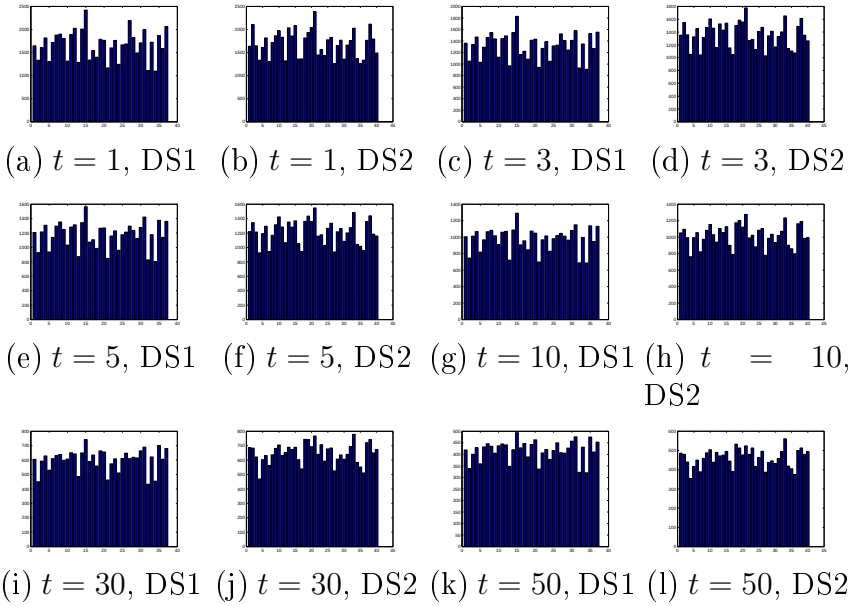
(i) $t = 30$, DS1 (j) $t = 30$, DS2 (k) $t = 50$, DS1 (l) $t = 50$, DS2

Figure 6.22: **Thresholding by term frequency** The histogram shows the distribution of how many senses a word belongs to (i.e., appears in the context of an example that is labelled with the sense). The threshold is denoted by t. It can be seen that as we increase the threshold, less and less words remain that could discriminate between senses.

increase the threshold, the discriminating words remain thanks to Inverse Sense Frequency.

Fig. 6.27 shows how many words the examples for each of the senses contain. The number of words (features) do not decrease significantly for any of the senses as the threshold grows.

In conclusion, TFISF keeps the more discriminating features, but does not reduce sparsity as well as thresholding by term frequency.



(a) $t = 1$, DS1 (b) $t = 1$, DS2 (c) $t = 3$, DS1 (d) $t = 3$, DS2

(e) $t = 5$, DS1 (f) $t = 5$, DS2 (g) $t = 10$, DS1 (h) $t = 10$, DS2

(i) $t = 30$, DS1 (j) $t = 30$, DS2 (k) $t = 50$, DS1 (l) $t = 50$, DS2

Figure 6.23: **Thresholding by term frequency** The figure shows how many words the examples for each of the senses contain. The senses are on the horizontal axis, and the number of words that appear in all the examples labeled with that sense are on the vertical axis. The threshold is denoted by t. The number of words do not decrease significantly for any of the senses as the threshold grows.

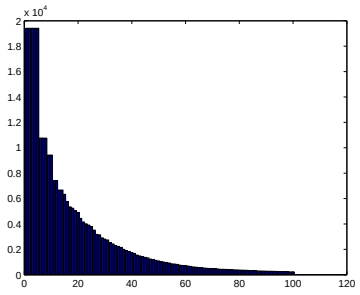


Figure 6.24: **Thresholding by TFISF** The figure shows the number of words that remain in the dataset as the threshold grows from 0 to 100. The threshold is on the horizontal axis, and the number of words is on the vertical axis.

word	examples	senses	correct		precision	
			BOW	ESA	BOW	ESA
promise	2012	19	959	964	0.477	0.479
band	1325	24	991	988	0.748	0.746
accident	1233	8	907	876	0.736	0.710
behaviour	993	3	953	953	0.960	0.960
shake	962	30	508	457	0.528	0.475
giant	658	14	282	321	0.429	0.488
shirt	532	9	343	303	0.645	0.570
generous	510	6	129	117	0.423	0.382
brilliant	442	13	206	196	0.466	0.443
knee	434	23	241	241	0.555	0.555
slight	384	8	347	327	0.904	0.852
modest	373	9	232	229	0.622	0.614
wooden	361	5	344	344	0.952	0.953
amaze	315	2	224	208	0.711	0.660
bother	293	12	162	162	0.552	0.553
seize	291	14	124	119	0.426	0.409
sack	285	10	184	182	0.646	0.639
float	282	26	59	67	0.209	0.236
bury	271	12	103	97	0.380	0.358
derive	258	7	101	107	0.391	0.415
excess	250	9	124	127	0.496	0.508
calculate	248	7	179	179	0.722	0.722
bet	165	17	60	54	0.364	0.327
bitter	143	11	43	51	0.301	0.357
sanction	95	15	53	52	0.558	0.547
consume	66	7	32	25	0.485	0.379
scrap	56	11	31	27	0.554	0.482
invade	53	8	21	21	0.396	0.396
onion	25	2	23	23	0.920	0.920

Table 6.2: Results of kNN-based word sense disambiguation on the *test* dataset of the SENSEVAL-1 English lexical sample task, with k=25, and with the context radius of 7. The ESA-based results show no improvement over the BOW-based ones. The mean accuracies were: 0.608 (BOW) versus 0.596 (ESA).

Dataset name	Senses	Examples per sense	Width of context
DS1	37	50	50
DS2	40	50	50
DS3	37	50	500

Table 6.3: **The data sets used**
There are three different datasets. DS1 and DS2 are very similar, but contain different senses. DS3 is the same as DS1, but the context is ten times as wide.

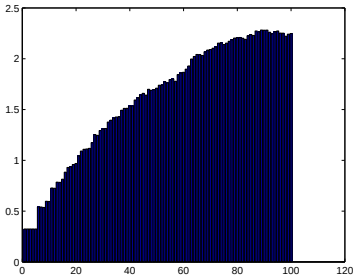
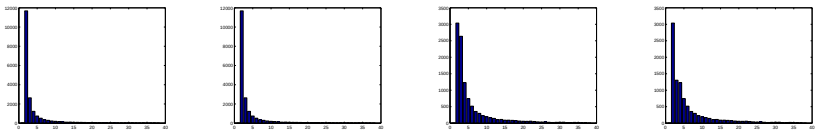
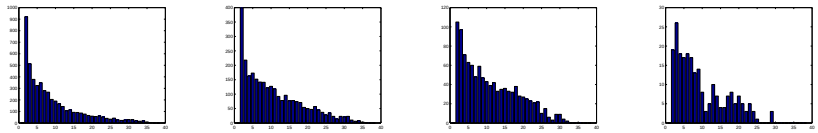


Figure 6.25: **Thresholding by TFISF** The figure shows the percentage of nonzero elements in the matrix A as the threshold grows. The threshold is on the horizontal axis, and the percentage of nonzero elements is on the vertical axis. It can be seen that the sparsity of the matrix can not be reduced as well as by thresholding by term frequency.

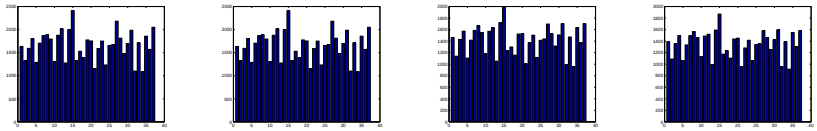


(a) $t = 0$ (b) $t = 5$ (c) $t = 6$ (d) $t = 9$

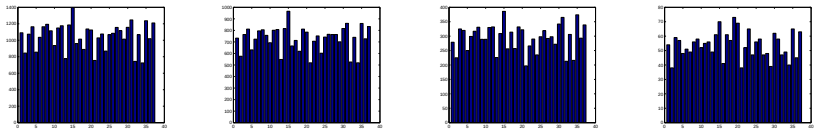


(e) $t = 20$ (f) $t = 30$ (g) $t = 50$ (h) $t = 100$

Figure 6.26: **Thresholding by TFISF** The histogram shows the distribution of how many senses a word belongs to (i.e., appears in the context of an example that is labelled with the sense). The threshold is denoted by t. It can be seen that as we increase the threshold, the discriminating words remain thanks to Inverse Sense Frequency.



(a) $t = 0$ (b) $t = 5$ (c) $t = 6$ (d) $t = 9$



(e) $t = 20$ (f) $t = 30$ (g) $t = 50$ (h) $t = 100$

Figure 6.27: **Thresholding by TFISF** The figure shows how many words the examples for each of the senses contain. The senses are on the horizontal axis, and the number of words that appear in all the examples labeled with that sense are on the vertical axis. The threshold is denoted by t. The number of words do not decrease significantly for any of the senses as the threshold grows.

6.4.4 Subspace Pursuit with Nearest Subspace

This algorithm is based on the Sparse Recognition and Classification [64] algorithm that was very successfully applied to face recognition. The algorithm consists of two steps. In the first step, we use Subspace Pursuit [15] to solve the following problem:

$$\min ||\boldsymbol{x}||_1 \text{ subject to } A\boldsymbol{x} = \boldsymbol{y} \tag{6.3}$$

In the language of compressed sensing, $\boldsymbol{y} \in \mathbb{R}^m$ is the measurement via the sampling matrix $A \in \mathbb{R}^{m \times n}$, and we want to recover the unknown sparse signal $\boldsymbol{x} \in \mathbb{R}^n$. In the language of natural language processing, $\boldsymbol{y}$ is a context, and the problem is to determine the meaning of this context. The columns of A contain contexts labeled with the sense they belong to. Our question is the following: What contexts (i.e., columns of A) should be combined linearly to get the context y? We try to achieve

$$\boldsymbol{y} = \boldsymbol{x}_1 A_1 + \boldsymbol{x}_2 A_2 + \ldots + \boldsymbol{x}_n A_n \tag{6.4}$$

with as few nonzero coefficients in $\boldsymbol{x}$ as possible. The sparsity constraint enforces that we obtain the context y from the linear combination of the least amount of contexts possible.

In the second step, we obtain the sense vector $\boldsymbol{s}$ based on how well the coefficients in $\boldsymbol{x}$ assiciated with each sense reproduce $\boldsymbol{y}$. So, for example, $\boldsymbol{s}_1$ is obtained by setting all the coefficients in $\boldsymbol{x}$ not associated with the first sense to zero, then computing $||\boldsymbol{y} - A\boldsymbol{x}||_2$. If we let $\delta_i : \mathbb{R}^n \to \mathbb{R}^n$ be a characteristic function that selects the coefficients of the ith sense, then

$$\boldsymbol{s}_i = ||\boldsymbol{y} - A\delta_i(x)||_2 \tag{6.5}$$

Algorithm 4 (Subspace Pursuit with Nearest Subspace)

1: **Input:** A matrix of contexts $A = [A_1, A_2, ..., A_n] \in \mathbb{R}^{m \times n}$ with each column labeled with one of k senses, and a context $\boldsymbol{y} \in \mathbb{R}^m$ to be interpreted.
2: Solve the following l_1 minimalization problem with Subspace Pursuit

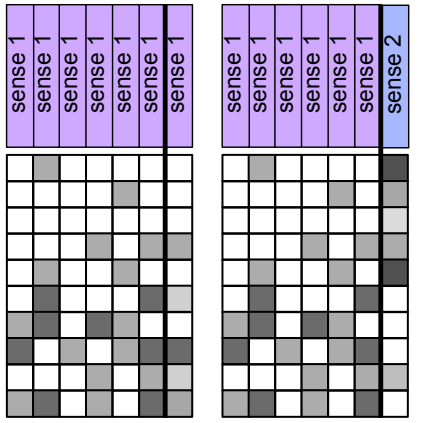
$$\min ||\boldsymbol{x}||_1 \text{ subject to } A\boldsymbol{x} = \boldsymbol{y} \tag{6.6}$$

3: **for** i = 1, 2, ..., k **do**
4: $\quad \boldsymbol{s}_i = ||\boldsymbol{y} - A\delta_i(x)||_2$
5: **end for**
6: **Output:** The sense vector $\boldsymbol{s}$.

6.4.5 Robust Principal Component Analysis based sense vector generation

This algorithm is based on Robust Principal Component Analysis (Sec. 6.4.7. In RPCA, we decompose A into two components $A = L + S$, where L has low-rank and the sparse S contains the outliers.

The main idea of the algorithm is that if we put the training examples of sense i into the columns of a matrix M together with a test example of the same sense, and we run the RPCA algorithm on $M = L + S$, then L will contain more of the energy of the text example than if we would put into M the training examples of sense i and a test example that belongs to a different sense. (Fig. 6.28) We can also say that less energy is absorbed from the test example. The reason is that when the test and training examples belong to the same sense, they are more correlated, so less of the test example is an outlier for RPCA.



(a) same sense (b) different sense

Figure 6.28: **RPCA based sense vector generation** Both of the subfigures show the M matrix. In the first case, the sense of the training examples are the same as the sense of the test example. In the second case, their senses are different. So, after running RPCA on both matrices, L will contain more energy of the test example in the first case.

In general, the more similar the senses of the training examples and the test example are in meaning, the more the energy that goes into L. Let $\delta_i : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}m \times n_i$ select the columns of the matrix A that contain the contexts of the ith sense. Then the ith coordinate of the sense vector is computed as follows.

First, we determine the matrix M.

$$M = [\delta_i(A)\, \boldsymbol{y}] \tag{6.7}$$

Then, we decompose this matrix into a low-rank L and a sparse S component using RPCA.

$$\min \|L\|_* + \lambda \|S\|_1 \text{ subject to } M = L + S \tag{6.8}$$

The ith component of $\boldsymbol{s}$ equals the energy of $\boldsymbol{y}$ that S contains. If l is the index

of the last column of M (i.e., the column that contains $\boldsymbol{y}$), then

$$\boldsymbol{s}_i = ||L_l||_2 \tag{6.9}$$

As an alternative method, we can also view the amount of energy that goes into S. In that case, the more similar the sense are, the less the energy in S.

Algorithm 5 (Robust Principal Component Analysis based sense vector generation)

1: **Input:** A matrix of contexts $A = [A_1, A_2, ..., A_n] \in \mathbb{R}^{m \times n}$ with each column labeled with one of k senses, and a context $\boldsymbol{y} \in \mathbb{R}^m$ to be interpreted.
2: **for** i = 1, 2, ..., k **do**
3: $M = [\delta_i(A)\ \boldsymbol{y}]$
4: Solve the following minimalization problem

$$\min \|L\|_* + \lambda \|S\|_1 \text{ subject to } M = L + S \tag{6.10}$$

5: $\boldsymbol{s}_i = ||L_l||_2$
6: **end for**
7: **Output:** The sense vector $\boldsymbol{s}$.

6.4.6 Results

We compare the accuracy of our methods and the baselines on the generalized WSD problem, using dataset DS1 and DS3. The sense assigned to each word is the sense with the largest coefficient in the sense vector. In the Subspace Pursuit with Nearest Subspace algorithm, we set K to 10.

As baselines, we use two linear multi-class Support Vector Machines. Support vector machines are inherently binary classifiers. To deal with multi-class problems (when there are M different classes), there are multiple approaches to combine multiple binary classifiers into a single multi-class one.

The one-against-all method combines M binary classifiers, one for each class. Every classifier decides one class and the rest of the classes. The outputs of the classifiers are then combined using a winner takes all strategy.

In the one-against-one method, a binary classifier is trained for each pair of classes. That means training $M(M-1)/2$ binary classifiers, so the number of classifiers is higher than in the one-against-all method, but the number of training examples is much lower in each classifier, which yields a better training performance. For a test example, the binary classifiers "vote" for a class. We use the libsvm [10] library.

Table 6.4 compares Subspace Pursuit with Nearest Subspace with the baselines. The one-against-all SVM has generally poor performance. When we threshold by word frequency, the one-against-one SVM approach is slightly superior to SPNS. When the matrix contains TFISF values, SPNS outperforms all the other methods at the low and middle thresholds. At the high thresholds, the number of features are significantly reduced.

Feature selection	SPNS	SVM-oaa	SVM-oao
no thresholding	76%	65%	78%
word frequency, threshold = 3	73%	65%	78%
word frequency, threshold = 10	76%	65%	76%
word frequency, threshold = 50	68%	65%	68%
tfisf, threshold = 0	**81**%	57%	76%
tfisf, threshold = 6	**81**%	57%	78%
tfisf, threshold = 9	**81**%	59%	78%
tfisf, threshold = 20	70%	59%	76%
tfisf, threshold = 50	65%	62%	75%

Table 6.4: **The results on the DS1 dataset** The three algorithms we compare are Subspace Pursuit with Nearest Subspace (SPNS), Support Vector Machine: one-against-all (SVM-oaa), and Support Vector Machine: one-against-one (SVM-oao). The one-against-all SVM has generally poor performance. When we threshold by word frequency, the one-against-one SVM approach is slightly superior to SPNS. When the matrix contains TFISF values, SPNS outperforms all the other methods at the low and middle thresholds. At the high thresholds, the number of features are significantly reduced.

We have introduced the DS3 database specifically to test the accuracy of RPCA based sense vector generation, as it performs better with matrices that are more dense. Here we only threshold the matrix by word frequency, as thresholding by tfisf does not reduce the sparsity as well (see figures 6.21 and 6.25).

Method	Accuracy
RPCA based 1.	75%
RPCA based 2.	75%
RPCA based 3.	75%
RPCA based 4.	**81**%
RPCA based 5.	78%
SVM-oaa	65%
SVM-oao	75%

Table 6.5: **RPCA based sense vector generation on the DS3 dataset** The RPCA-based methods are numbered according to the enumeration above. The RPCA-based methods outperform the other two methods.

After many experiments, we have chosen our threshold to be 10. Choosing a lower threshold makes the matrix more sparse, and higher thresholds discard too much information. We are testing five different versions of the algorithm. λ was set to 0.1.

1. We choose the sense with the smallest $||S_t||_2$ value.
2. We choose the sense with the largest $||L_t||_2$ value.

3. We use all the $||S_t||_2$ values as features in the one-against-one Support Vector Machine.

4. We use all the $||L_t||_2$ values as features in the one-against-one Support Vector Machine.

5. We use both the $||S_t||_2$ and the $||L_t||_2$ values as features in the Support Vector Machine.

where t is the index of the test example in the matrix $M = L + S$, L denotes the low-rank matrix, and S the outlier matrix.

Table 6.5 shows that the RPCA-based methods outperform the other two methods.

6.4.7 Robust Principal Subspace Analysis

Robust Principal Component Analysis

The original formulation of the Robust Principal Component Analysis (RPCA) is as follows [8]:

Suppose we are given a large data matrix M, and know that it may be decomposed as

$$M = L_0 + S_0,$$

where L_0 has low-rank and S_0 is sparse; here, both components are of arbitrary magnitude. We do not know the low-dimensional column and row space of L_0, not even their dimension. Similarly, we do not know the locations of the nonzero entries of S_0, not even how many there are. Can we hope to recover the low-rank and sparse components both accurately (perhaps even exactly) and efficiently?

The intriguing answer is yes, although at first sight, the separation problem seems impossible to solve since the number of unknowns to infer for L_0 and S_0 is twice as many as the given measurements in $M \in \mathbb{R}^{n_1 \times n_2}$. Furthermore, it seems even more daunting that we expect to reliably obtain the low-rank matrix L_0 with errors in S_0 of arbitrarily large magnitude.

The augmented Lagrange multiplier (ALM) method operates on the *augmented Lagrangian*

$$l(L, S, Y) = \|L\|_* + \lambda\|S\|_1 + < Y, M - L - S > + \frac{\mu}{2}\|M - L - S\|_F^2 \qquad (6.11)$$

A generic Lagrange multiplier algorithm would solve Principal Component Pursuit (PCP) by repeatedly setting $(L_k, S_k) = \arg\min_{L,S} l(L, S, Y_k)$, and then updating the Lagrange multiplier matrix via $Y_{k+1} = Y_k + \mu(M - L_k - S_k)$.

For our low-rank and sparse decomposition problem, we can avoid having to solve a sequence of convex programs by recognizing that $\min_L l(L, S, Y)$ and $\min_S l(L, S, Y)$ both have very simple and efficient solutions. Let $\mathcal{S}_\tau : \mathbb{R} \to \mathbb{R}$ denote the shrinkage operator $\mathcal{S}_\tau[x] = \text{sgnsgn}(x)\max(|x| - \tau, 0)$, and extend it to matrices by applying it to each element. It is easy to show that

$$\arg\min_S l(L, S, Y) = \mathcal{S}_{\lambda\mu}(M - L + \mu^{-1}Y). \qquad (6.12)$$

Similarly, for matrices X, let $\mathcal{D}_\tau(X)$ denote the singular value thresholding operator given by $\mathcal{D}_\tau(X) = U\mathcal{S}_\tau(\Sigma)V^*$, where $X = U\Sigma V^*$ is any singular value decomposition. It is not difficult to show that

$$\arg\min_L l(L, S, Y) = \mathcal{D}_\mu(M - S - \mu^{-1}Y). \tag{6.13}$$

Thus, a more practical strategy is to first minimize l with respect to L (fixing S), then minimize l with respect to S (fixing L), and then finally update the Lagrange multiplier matrix Y based on the residual $M - L - S$, a strategy that is summarized as Algorithm 6 below.

Algorithm 6 (Principal Component Pursuit by Alternating Directions)

1: **initialize:** $S_0 = Y_0 = 0, \mu > 0$.
2: **while** not converged **do**
3: compute $L_{k+1} = \mathcal{D}_\mu(M - S_k - \mu^{-1}Y_k)$;
4: compute $S_{k+1} = \mathcal{S}_{\lambda\mu}(M - L_{k+1} + \mu^{-1}Y_k)$;
5: compute $Y_{k+1} = Y_k + \mu(M - L_{k+1} - S_{k+1})$;
6: **end while**
7: **output:** L, S.

Upgrading RPCA to an online algorithm: the Robust Principal Subspace Analysis

We can not use RPCA for our huge databases. Known single pass PCA methods could be used here, but for our case an easier method is sufficient, because we do not need the order of the principal components. Instead, we need only the subspace of RPCA and to filter out the outliers from that subspace. This is big difference. For example, if the database changes and the magnitudes of the singular values of the n^{th} and m^{th} principal components exchanges, then re-computation of PCA becomes necessary. On the other hand, the principal subspace remains the same. This also means that the complexity of the pursuit is less. Online learning rules exists for Principal Subspace Analysis [50] and its robust version (RPSA) provides the same results as RPCA according to our numerical experiments.

The RPCA and RPSA method separates the typical (L) and the outlier/sparse (S) part of the data. The typical part can be filled in even in case if a large portion of the information is missing. The main advantage of RPCA/RPSA is that they sparsify the coordinates of the dictionary without thresholding. Thresholding would spoil dictionary elements as one can easily see it for overcomplete wavelet dictionaries.

Chapter 7

Progress in scaling up the project

7.1 Design of a new architecture

In this semester, we have worked out the details of our project. We generate novelties in three phases (Fig. 7.1). First, we collect and store topical content from all over the internet. This task is solved by our crawler system developed in the previous semester. Then we retrieve a number of candidate text fragments to work on with our deep techniques. This second step requires efficient data storage and retrieval capabilities, for millions of documents. In the third step, we use our deep techniques to generate novelties.

We have created a new software architecture to integrate these steps based on Lucene, the open source informational retrieval software, and Nutch, the open source crawler based on Lucene (see the Appendix for details on software). For the first step, the crawled pages are parsed and then stored in a data structure (an extended inverted index) that retains and makes queriable all the information generated (e.g., terms, dependency relations, part of speech, senses). In the second step, we query this data structure using the capabilities of Lucene to get the candidate text fragments we want.

In the third step, cascades are generated using the link database that stores the incoming links for each document. The architecture also helps the work with senses by storing and retrieving the training examples for word sense disambiguation, and by storing the senses of disambiguated words to make retrieval based on senses possible.

The structure of this section reflects the three main steps described above. Each step was given its own subsection.

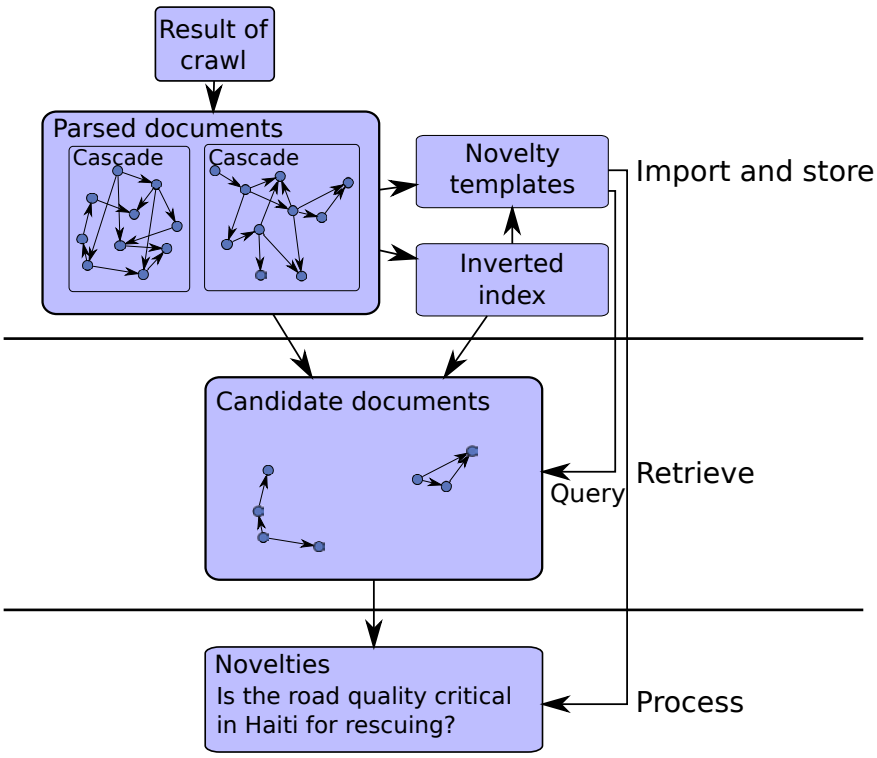


Figure 7.1: **The process of novelty generation** We generate novelties in three phases. First, we collect and store topical content from Blogspace. Then we retrieve a number of candidate text fragments based on novelty templates. In the third step, we use our deep techniques on the candidate documents to generate novelties. Novelty templates are generated based on information cascades (See Sec. 7.1.3). The process of question answering is very similar, except that the question is used to retrieve candidate text fragments, and the deep techniques are different (see results in the previous semester).

7.1.1 Storing information

We collect lots of information in the form of HTML documents from the WWW. In addition, as we parse these documents, various kinds of additional data are generated (e.g., part-of-speech, dependency relations, word senses) that needs to be stored. We have designed a data structure that makes storing and retrieving this data efficient.

The data structure

We have realized a data structure based on the inverted index (Sec. 7.1.2) of Lucene that integrates all our data structures used previously (e.g., dependency graphs, part-of-speech tagged text, raw text, sensegraphs, cascades, etc). These previous data structures are now stored in our combined data structure.

The basic unit we store and index is the document. We build a corpus where each document is indexed. The various data (described above) are stored for each term in each document.

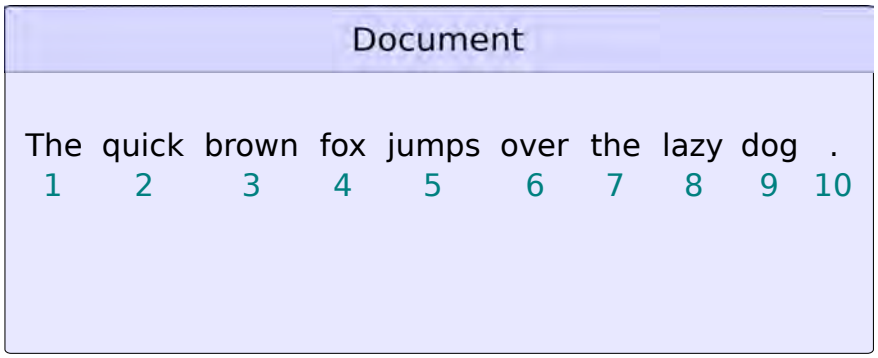


Figure 7.2: **Data structure - the raw text** The text of the document supplemented with the position of each term. In the example, the document consists of only one sentence.

We describe the structure of a document step-by-step: we start with raw (tokenized) text, and add the new layers of information one-by-one. Fig. 7.2 shows the text supplemented with the position of each term. On Fig. 7.3, we added the part-of-speech tag to each term. Lucene allows us to do this by attaching so-called *payloads* to each term. The payloads can be used in queries. For example, we can collect all the documents where the word `fox` is a noun.

On Fig. 7.4 dependency relations have been added to each sentence. In dependency parsing, each dependent has only one head, so we simply store the head of each term, and the type of the dependency relation. In our example, there is a dependency relation between the head word `fox` (on position 4) and the dependent word `The` (on position 1) of the type *determiner*. We store this relation by adding the pair (4, *determiner*) to the word `The` (on position 1). The root of the dependency tree has no head.

On the next figure, Fig. 7.5, the sentence boundaries are marked, so we can query sentences (e.g., for question answering). Paragraphs and sections can also be marked and used as contexts.

Fig. 7.6 contains also senses of words from manually tagged corpora (i.e., training examples for word sense disambiguation). This allows us to handle word sense disambiguation in the same context we handle the other tasks, and to integrate various knowledge sources (e.g., Wikipedia, the corpus downloaded

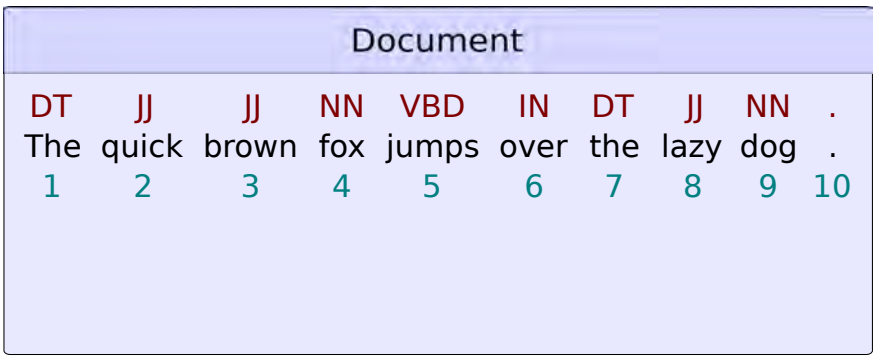


Figure 7.3: **Data structure - with POS tags** The part-of-speech tag of each term is also stored and indexed.

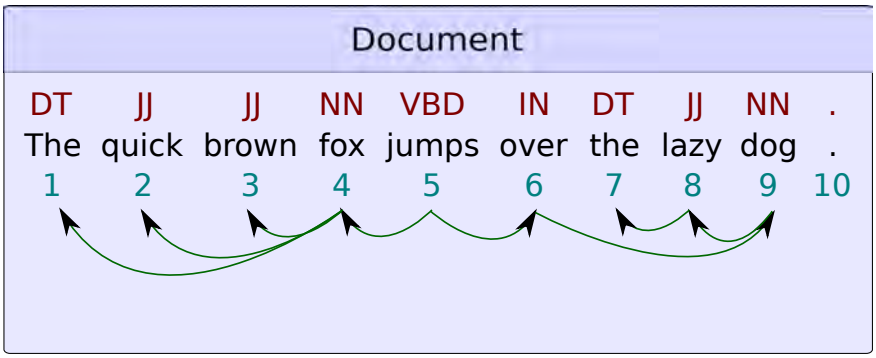


Figure 7.4: **Data structure - with dependency relations** In addition to the POS tags, dependency relations have been added to each sentence.

from the WWW). In the example, the senses are from Wikipedia, as in [45]. In general, different sense inventories can be used, for example, another document can be annotated with WordNet senses. The contexts of the various senses of a word (e.g., for using them as training examples) can be easily collected by queries.

Lucene allows us to store several terms at the same term position in the index. We use this feature to assign senses to every term. Where we have manually tagged sense, we only duplicate it (i.e., store it also in the term position besides as payload). For the rest of the terms, we can obtain senses by automatically disambiguating using word sense disambiguation (Fig. 7.7. This allows us to use senses instead of words in our algorithms without modifications.

Cascades are stored in the link database of Nutch. Incoming links and hostnames are stored for each document. Identifying cascades is easy based on

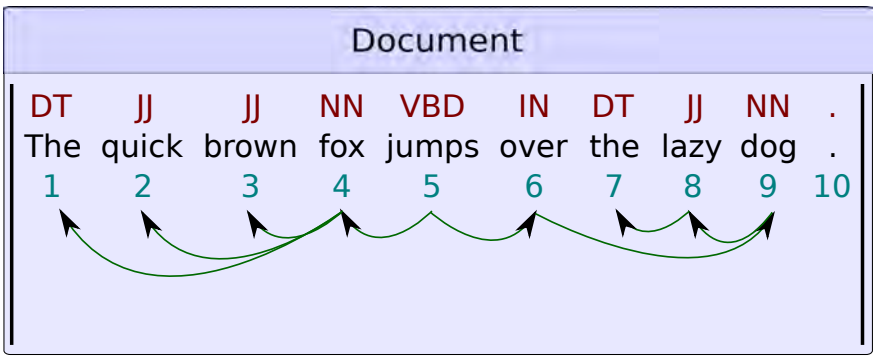


Figure 7.5: **Data structure - boundaries** Sentence boundaries are marked, so the whole sentence that contains the queried terms can be retrieved.

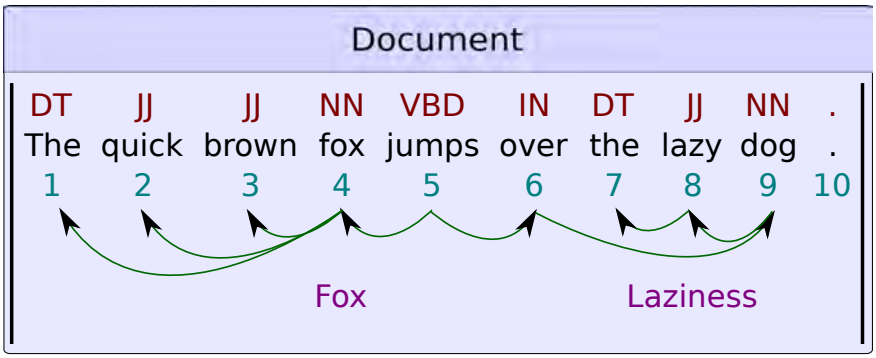


Figure 7.6: **Data structure - manually tagged senses** Senses from manually tagged corpora are stored for each term where available.

this information. For details, see the next section.

7.1.2 Retrieving text fragments

The first step in our three step process is retrieving some candidate text fragments to work on with deeper, slower methods (e.g., graph kernels). Fast information retrieval is made possible by *indexing* the corpus.

Indexing and querying a corpus

Here we describe the indexing and querying of a corpus using an information retrieval system in general.

To avoid parsing through the whole corpus each time a query is invoked, an *index* must be created. An index is a data structure which enables quick lookup

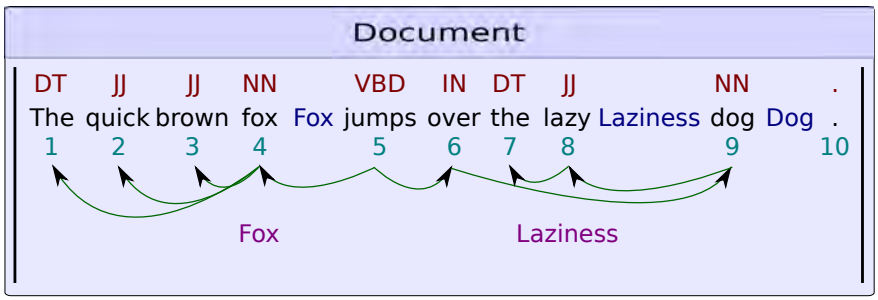


Figure 7.7: **Data structure - senses** The senses of each term (manually tagged or automatically disambiguated) are stored in the index, so we can query senses instead of terms.

of content.

Indexing a corpus has multiple phases. The first of these is *text extraction*, the extraction of terms from a document. It starts with parsing and tokenizing the document, recovering the individual words of it. All formatting and punctuation are discarded. Usually, indexing all terms is unnecessary: words like 'the' or 'and' do not relate to the topic of the document. A *stopword list* contains the list of words we do not want to index altogether.

Most languages have multiple morphological variants of the same word, which are closely related (e.g., 'fishing', 'fished', 'fish' etc.). A *stemmer* is usually used to strip the inflectional suffixes from all words, to bring these variants of the words to the same form. Note that this step does not necessarily produce real words (e.g., it is considered to be acceptable if the words 'house', 'houses' and 'housing' are brought to the form 'hous'), as the result of text extraction is not directly shown to the user. Another way to improve the quality and performance of the index to impose a *minimum term frequency* restriction, i.e., to skip over terms that appear in a document fewer times than specified.

The relevant terms of documents are indexed in a way which promotes the fast retrieval of the documents that belong to these terms. Most information retrieval systems use a so-called *inverted index* (Fig. 7.8). It is called like that, because instead of storing each word to a document (which is essentially the document itself), they store each *document* to a term. It is a table of terms, which maps from them to the documents they occur in. To support *phrase* or *proximity* searching, the notion of inverted index can be extended to store the positions for each occurrence of the term as well.

Using an information retrieval system involves the formulating of queries. The system processes these queries, and provides the results in some way. The queries are usually described in a formal language, and can be single words to search for, as well as complex queries using search for phrases (where the specified terms have to appear in consecutive order) or proximity search (where the terms have to appear in close linear distance to each other). Some informa-

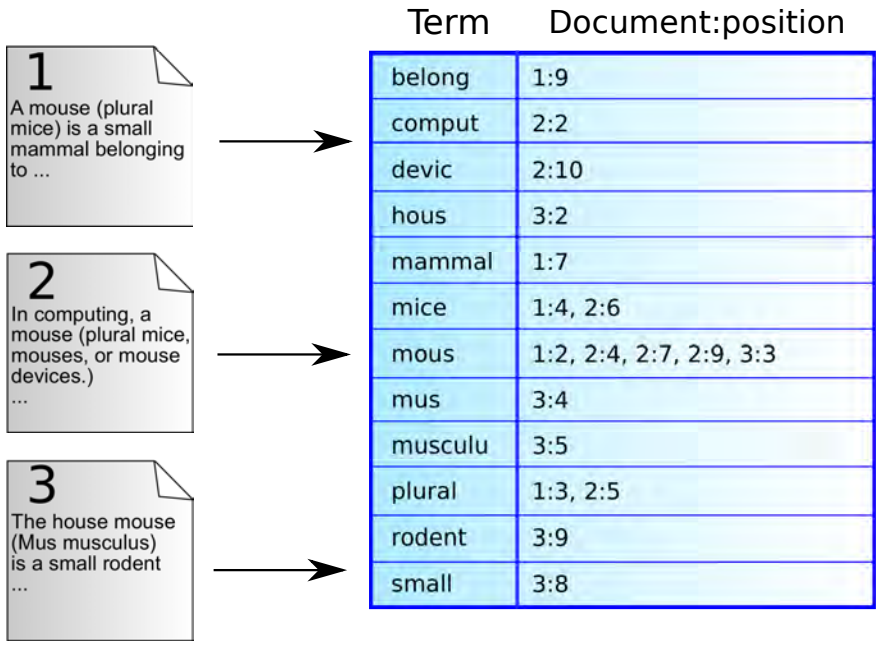


Figure 7.8: **The inverted index** An inverted index maps terms to documents they occur in. It makes retrieving documents that contain specific terms efficient. On the figure, we index three documents. There is a list of document:position pairs for each term. For example, if we would like to retrieve all the documents that contain the term "mice", we would only have to look up the term "mice" in the inverted index, and return the first two documents.

tion retrieval systems provide additional support for excluding documents with specific properties (such as date).

The results of information retrieval are the documents in the corpus. They are presented to the user sorted by their supposed relevance. A trivial way to estimate relevance is to take the *term frequency* in the document into account. This can be refined in a number of ways.

Retrieval for various tasks

Now, after introducing indexing and retrieval in general, we show how the different aspects of the project are using information retrieval. All of these applications make use of the inverted index that allows us to quickly retrieve the documents where some word, phrase, word sense, etc. occurs.

Dependency graphs and SenseGraphs

We can produce dependency graphs or SenseGraphs of sentences, paragraphs, sections or whole documents in very little time. We simply choose the part of the text we want to create the graph from, and construct the graph from the individual parsed sentences. We iterate over all sentences, and add the stored dependency relations as edges to the graph. In SenseGraphs, we add senses instead of words to the graph as nodes.

Question answering and analogy detection

Question answering and analogy detection are both two-step processes. First, we collect a number of text fragments (e.g., sentences, paragraphs, or whole documents) using fast information retrieval techniques. Then we process these smaller text fragment collections using slower, deeper techniques (e.g., graph kernels). A huge improvement over the previous, hierarchical system (i.e., the one we designed in the previous semester) is that we can retrieve the sentences directly, that is, we do not have to work hierarchically.

The task of question answering and retrieving documents that satisfy an analogy are very similar. In question answering, we query sentences that may contain candidate answers. Finding documents that satisfy an analogy involves finding text fragments that are similar to the representation of the analogy (Sec. 7.1.3). Because we have to retrieve text fragments similar to another text fragment in both tasks, we can discuss them together.

We can retrieve text fragments based on words, senses, part-of-speech tags, dependency relations, or any combination of these. All of this information is useful to reduce ambiguity. For example, if the question is `Who cast the sword in the film?`, knowing the sense, the part-of-speech, or the dependency relation of the word `cast` can all serve to disambiguate the query. Without these, querying a film database would return casts of films, instead of films in which weapons were cast.

Phrases can be queried using Lucene SpanQueries. For example, if we want to find the phrase `little house`, we can query `house` following `little`, with `little` within 2 positions of `house`. So the query returns, for example, `little white house` in addition to `little house`. These queries can be combined, so more complex queries can be created like `little house`, but without `green`, or `little house` followed by `build`.

Cascades

As the incoming links and hostnames are stored for each document, identifying cascades is easy and fast. An additional possibility is extending the link database with 'missing' links. If two documents are very similar, but there is no link between them, then information has spread from one to the other, but without identifying the source. We add a new edge to the graph of information spreading between these similar documents. This is feasible because we can retrieve documents that are similar to a document quickly.

Word Sense Disambiguation

The data structure also helps word sense disambiguation. Collecting training examples for a word or a specific sense of a word is done by querying the corpus. Queries can take into account the part-of-speech, the dependency relation, the document title, etc. These can all be used for word sense disambiguation. For example, we can take into account the part-of-speech of a word. Some words can be disambiguated based on that information alone. The data structure detailed in Sec. 6.2.4 can be built on a word-by-word basis.

Two new possibilities that arise here are the following. First, we can find links to wikipedia pages in non-wikipedia documents. If we find such a link, we can consider the anchor text disambiguated. Second, we can increase the number of our training examples as detailed in Sec. 6.2.4, but for the whole corpus. We find all the anchor texts which link to only one Wikipedia page (i.e., they are monosemous), and then collect contexts of these anchor texts from the non-wikipedia corpus as well.

7.1.3 Processing information - cascades

Cascades are currently used for two purposes. For measuring the spreading of information, we have to monitor a number of blogs. We use algorithms based on submodularity that take the information cascades as input to select these blogs. On the other hand, we also use cascades to collect analogies. In this semester, we have worked on both tasks.

Experiment with cascades based on page content

In the last semester, we have experimented with information cascades based on the work of Leskovec et. al. [41]. Leskovec has taken granted that if a blog post links to another post, then information has spread between the two posts. We would like to test this assumption by measuring the similarity of the posts with a link between them, and then thresholding the graph of information spreading (i.e., removing all the edges where the similarity between two posts is less than the threshold) before running the algorithm of Leskovec et. al.

This way we will obtain information cascades where information has certainly spread between the blog posts in the cascade. If we do not perform this thresholding, it may be the case that the two blogs are linked only because, for example, one has a link on its menu bar to the other.

In this semester, we have performed a large crawl to collect blog posts and the link structure for this experiment. We have performed the experiment on smaller data sets. However, we were not able to run it for the whole, large data set, due to technical difficulties (memory problems). We will resolve these problems in the following semester, and describe the results in our next report.

Finding analogies

For creating blog posts that spread, we combine different blog posts if similar information has been combined in the past, and has been successful (i.e., spread widely). The success of the information (blog post) can be approximated by the size of its information cascade.

Creating new information by combining existing information is more difficult. We do this by finding analogies: we find blog posts that were combined from different blog posts, and create our blog post in an analog fashion. So an analogy is basically a pattern that allows us to combine blog posts. We also use the term *novelty template* for analogy, because novelties are generated based on these patterns.

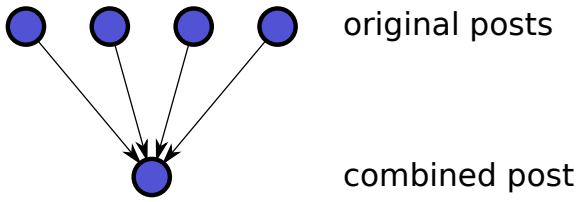


Figure 7.9: **Combining blog posts** A subset of a cascade where information has spread from the original posts and was combined in the combined post. We can find these subsets, and detect analogies.

One way to find such analogies is to look for them in the graph of information spreading (i.e., in the cascades). If, in this graph, information has spread from several different posts (called the original posts) into a single post (called the combined post), it is likely the case that the single post has been created by juxtaposing information from the original posts. We can detect this, and characterize analogies based on these patterns (Fig. 7.9).

In general, we want to identify for each original post the part it has spread to in the combined post. One solution is to take the original posts one-by-one, and look for the parts of the combined post that are similar to parts of the original post. For example, a simple detection mechanism would be to compare the Bag of Words representation of the original post with the combined post, and work with the words that appear in both. But it is hard to decide what is important in the intersection of the two Bag of Words': there will be reasonably common words that are nonetheless not important.

Additional information can be used to determine only the parts that are similar in the two posts because of information spreading. We can assume that sentences are the unit of information spreading, not mere words. So we want to find the sentences in two documents that are similar. As sentences are usually not taken from other posts verbatim, we have to account for changes in wording. This can be done with our graph kernel based sentence comparison methods that were developed in the previous semester.

The exact algorithm is the following. As inputs, we have several original documents, and one combined document. We want to determine the information that was spread from each original document to the combined document. So we take the original documents one-by-one, and create a bipartite graph, where the two independent vertex sets are the sentences of each document. We select for each sentence of the combined document all the sentences that are more similar to it in the original document than a predetermined threshold. This way, if several sentences were combined, they can be found.

After we have the text fragments that were spread from each of the original documents, we can retrieve similar text fragments and combine them to create novelties.

7.2 Development

7.2.1 Scaling up the project

Gathering data

In the previous semester, we had problems while collecting data from the Web using our crawler. The crawler we based our system on, Heritrix, was not capable of collecting more than approx. 120 gigabytes of data. At that point, it ran out of memory because of memory management issues, so it stopped. Because of this, we had planned to update the base of our crawler to Heritrix 3.0, an other branch of the software (`https://webarchive.jira.com/wiki/display/Heritrix/Release+Notes+-+3.0.0`). This change involved considerable amount of work that we did half way. Fortunately, the other half could be saved since then a new version came out from the branch we are using (Heritrix 1.14.4, `https://webarchive.jira.com/wiki/display/Heritrix/Release+Notes+-+1.14.4`) that solved this problem.[1]

We have updated the software to be based on Heritrix 1.14.4. Since then we could collect as many as approx. 250 gigabytes of data without problems (the crawl was stopped because we did not need more data).

Accessing and storing documents

In the previous semester, we have designed an architecture to store and access our documents efficiently. In the past, we have stored documents as separate files in the file system. This method became more and more inconvenient: as the number of documents grew, the feasibility of the studies were at risk given the speed limitations. We decided to store the files in some kind of database. The best solution was an information retrieval system as it solves not only the storing problem, but it alleviates the document accessing task in various ways by

[1] Quote from the new version: "A number of performance, memory-retention, and deadlock-risk issues occasionally affecting the implementation class CachedBdbMap were identified. Fixes have been applied, but also the class has been replaced with a more simple implementation focused specifically on Heritrix's common use cases, ObjectIdentityBdbCache."

means of the inverted index (Fig. 7.10). For example, we can collect instances of a word easily by querying that word. This is useful e.g., when collecting training examples for word sense disambiguation.

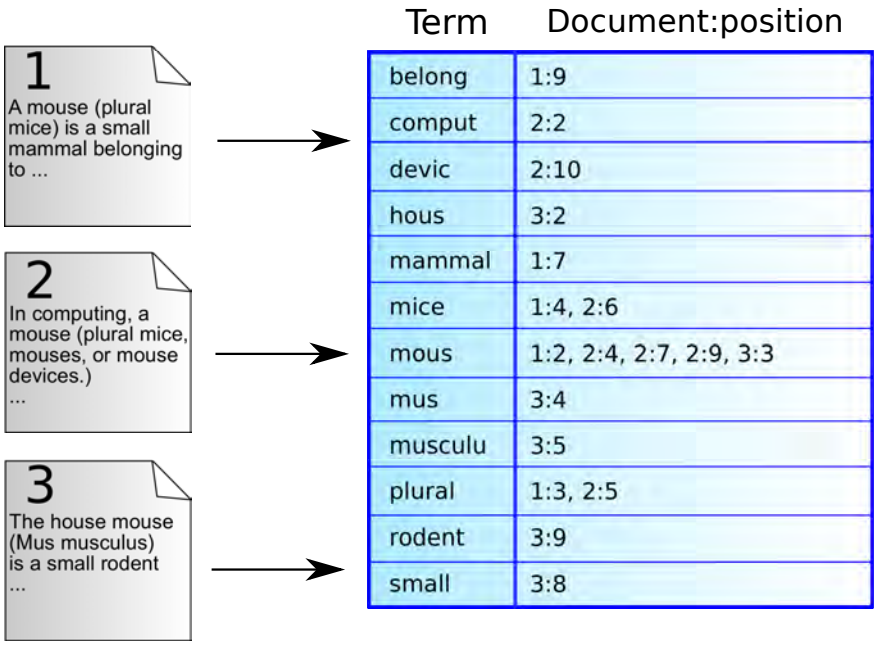


Figure 7.10: **The inverted index** An inverted index maps terms to documents they occur in. It makes retrieving documents that contain specific terms efficient. On the figure, we index three documents. There is a list of document:position pairs for each term. For example, if we would like to retrieve all the documents that contain the term "mice", we would only have to look up the term "mice" in the inverted index, and return the first two documents.

The other advantage of an information retrieval system is that additional data (e.g., the result of dependency parsing or Part-of-Speech tagging) can be also stored and accessed efficiently. Currently, we do not store this data because of the improved parsing speed makes this unnecessary, but the capability is there when we need it.

In this semester, we have implemented the software based on Lucene, an open-source information retrieval system (`http://lucene.apache.org/`) and based our tools on it. In the last semester, we also planned to include Nutch (`http://nutch.apache.org/`), but after some period of development, we decided against it, because of a number of problems we had to face and that we needed lower level access to the internal workings of Lucene; e.g., we needed term vectors.

Parsing documents

One significant bottleneck in scaling up the project was the parsing speed of our dependency parser, `MaltParser`. It could parse approx. 1 sentence per second, which is way too low for our purposes. There are many dependency parsers besides `MaltParser`, so we decided to gather as many open-source dependency parsers as we can, and tested their parsing speed. In the end, we found that `DeSR` (`http://desr.sourceforge.net/`) is one of the fastest parsers. `DeSR` is capable of parsing roughly 200 sentences per second on our computer (Core2Duo 2 GHz).

Meanwhile, a new version of `MaltParser` was made available that uses LIBLINEAR [16] (`http://www.csie.ntu.edu.tw/~cjlin/liblinear/`), a very fast classifier we use also in our own crawling system. We tested this version of `MaltParser`, and obtained comparable results for its speed as that of `DeSR`. Because `MaltParser` is written in `Java`, and we have already interfaced it to our software, we chosen to stay with it but upgraded it to the latest version. This new version of `MaltParser` is fast enough so we can parse documents on the fly when it is needed. We no longer need to parse beforehand, and store the parsed data for later retrieval.

7.2.2 Architecture for mining Wikipedia as a sense-annotated corpus

In the previous semester, we have designed an architecture to help interpret the meaning of texts. In this semester, we have implemented this architecture. It uses Lucene to index and retrieve articles and links of Wikipedia. The main advantage of the software is that handling of this very large dataset became easy and efficient. For example, we can harvest training examples for word sense disambiguation very quickly across the whole Wikipedia.

Information retrieval software

We have developed an information retrieval software for Wikipedia. It is based on the Apache Lucene library. It indexes and searches the individual words and links. Searches with complex queries can be performed in seconds. This software is essential to our future experiments on Wikipedia; now we do not have to to read through the whole database each time we want to retrieve data from it. That would take many hours.

Processing the link structure of Wikipedia is not a trivial task. The articles are identified by their title, but titles can be written in a couple of different forms. For example, it does not matter if the first character is in upper or lower case: the titles COMPUTER and COMPUTER refer to the same article, because the first letter is automatically converted to upper case (note that the cases of the other letters do matter, e.g., RED MEAT and RED MEAT are different article titles: the former is a type of meat, the latter is a comic strip). Another example: the individual words in the title can either be separated by spaces or

underscores, the result is the same.

The task of handling links is also troubled by special pages called *redirects*. These articles send the reader to another article, usually from an alternative title[2]. For example, the page UK is a redirect page. It does not have a content; instead, it contains a single link to the article UNITED KINGDOM. When browsing Wikipedia online, clicking to a link that is pointing to UK takes the reader to UNITED KINGDOM instantly.

This example raises another problem for us: the anchor text UK and the anchor text UNITED KINGDOM look the same from the point of view of Wikipedia, but seem to be different according to our experiences: such links (another example is CD and Compact Disk) might belong to different topics, where their contexts differ. Differences may arise from the descriptions provided for disk experts as opposed to music fans and it can confuse disambiguation unless we can distinguish between topics.

Figure 7.11: **Our software creates a Lucene index from an XML file.**

The indexing process works on the Wikipedia XML dump, which is an XML-file that contains all the articles of the English Wikipedia. Such dumps can be downloaded freely[3]. The indexing process consists of three runs, and includes the following steps:

1. During the first run redirects are extracted, to be stored in a hash table for the following steps of the process. As the English Wikipedia contains more than 3 million redirects, storing this hash table requires a considerable amount of computer memory. This step is necessary because a quite large portion of links in Wikipedia point to redirects, and the preliminary resolution of such redirects simplifies and speeds up searches.

2. During the second run, for each article, the number of inward links are counted (this information is not stored explicitly in the XML, it can be obtained only by counting the individual links). This data is stored, and later added to the resulting index, to enable the user to search only among those articles which have the specified number of links pointing to them. This is used to discard unimportant data.

[2] `http://en.wikipedia.org/wiki/Wikipedia:Redirects`
[3] `http://en.wikipedia.org/wiki/Wikipedia:Database_download`

3. The final run performs the actual indexing. For each article, the text of the article is extracted. Formatting characters and tags (e.g., apostrophes indicating emphasis, `<ref>` and `</ref>` tags indicating references) are dropped. The remaining text is then tokenized, indexed and stored. Links are also extracted, redirects and different spellings of the same sense (e.g., DEVELOPING_COUNTRY and DEVELOPING COUNTRY – notice the different capitalization and spacing) are resolved. Links are also indexed and stored. For each article, the following data is also stored and indexed: the number of inward links, the number of outward links, the number of words, and whether the page is a disambiguation page (these are pages that simply enumerate the different senses that belong to an ambiguous term – and thus do not contain enough meaningful text that could be used as a training example for word sense disambiguation).

The resulting index can be used to perform complex searches very fast. For example, finding all non-disambiguation articles that are at least 200 words long, have at least 5 inward links, and contain a link to MOUSE (COMPUTING) takes less than a second. More complex searches (such as finding second-order links – see later) require more queries, but even these can usually be performed in a minute.

Architecture for interpreting text

Following is a description of the system we have implemented. It incorporates knowledge from the link structure of Wikipedia and from the texts of the articles.

First, we create a data structure of many sense-tagged words with contexts. We treat the link structure of Wikipedia as a bipartite graph, where the edges represent the links. The elements of one vertex set are labeled with the anchor texts of the links. The elements of the other vertex set are labeled with the target article of the links. These are the *senses*, or *concepts* in the terminology of Wikipedia-based word sense disambiguation [22]. One link can be thought of as a sense-tagged example in a corpus. Our goal is to collect as many examples as possible.

To achieve this goal, we build a data structure in two steps: marking the links to gather the examples, and gathering the examples.

In the first step, we mark all the links in the bipartite graph that can be used for gathering examples for a given term (the method also works for phrases that consist of multiple terms). First, we take all the links whose anchor text is exactly the term (See Figure 7.12). We call these *first-order links* with respect to the term. These are obviously good sense-tagged examples. After that, we also mark all the links that point to some sense that some already marked link also points to (See Figure 7.13). We call these *second-order links* with respect to the original term, because we can only retrieve these on the basis of the first-order links. This way we will mark all the links that have a common sense with some link whose anchor text is exactly the term. We conjecture that these are good examples regardless of their anchor text, because the anchor text is used

in the same sense as the term we are collecting examples for. The advantage of this method is that we gather much more correct examples than with previous approaches.

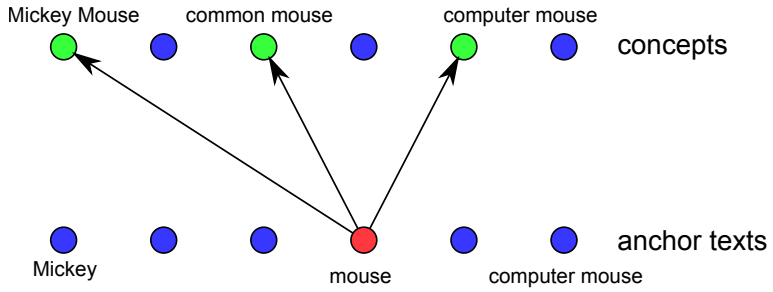


Figure 7.12: **Marking the first-order links for a term-concept data structure.** The dots in the bottom represent the anchor texts of the links. The target articles of the links (i.e., the senses or concepts) are on the top. For example, three links have the anchor text 'mouse', one of which points to the sense MICKEY MOUSE. For the term 'mouse' (marked with red), we mark the links that have the anchor text 'mouse' (concepts marked with green).

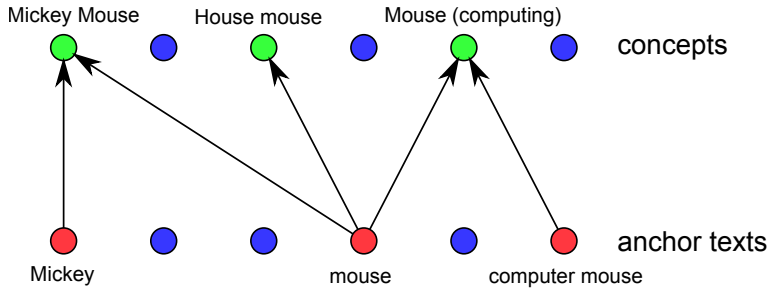


Figure 7.13: **Marking the second-order links for a term-concept data structure.** Continuing the marking the links to gather examples from (see Figure 7.12), we mark all the links that point to some concept some already marked link also points to (the concepts marked with green). For example, we collect the link with anchor text 'Mickey', because it points to a previously marked concept, namely MICKEY MOUSE.

In the second step, we gather the examples from the marked links. We see each link as an example for the given term and the sense it points to. We collect the contexts of the anchor texts of these links, and label them with the article that contained the link (See Figure 7.14).

The architecture has a number of advantages compared to our previous approaches. We expect it to resolve many of the problems we encountered. Explicit

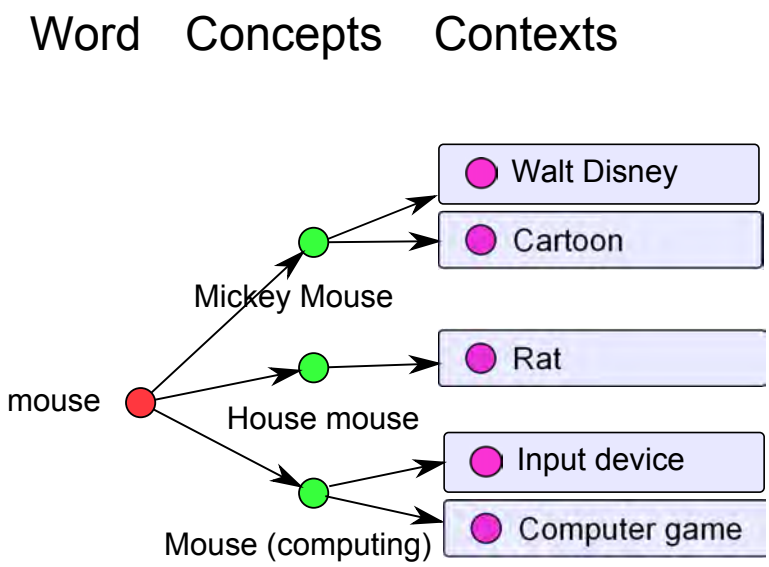


Figure 7.14: **The resulting term-concept data structure.** The term-concept data structure is created from the link structure seen in Figure 7.13. For each term, concepts are assigned, and for each concept, contexts are assigned. For example, the concept MICKEY MOUSE is assigned to the word 'mouse' (because there is a link in Wikipedia which points to MICKEY MOUSE, and whose anchor text contains 'mouse'), and contexts of anchor texts from articles WALT DISNEY and CARTOON are assigned to the concept MICKEY MOUSE (because these links point to MICKEY MOUSE).

Semantic Analysis [21] assigns an overly fine-grained concept representation of a word. Senses that links point to are much more sparse and more coarse. ESA assigns the same vector to a word regardless its actual sense. Our system will perform disambiguation based on the contexts of links in Wikipedia. We used links as training examples before, but the new architecture allows us to gather more. Finally, the concepts we will assign to words will be more relevant than the ones ESA assigns, because the links in Wikipedia denote keyphrases.

Results so far

We have conducted some experiments using the software. Our methodology was the following:

1. We chose ambiguous words randomly from Wikipedia. We found that the most interesting and hardest cases are the ones where there is a relatively few training examples, so most of the tests were done using only words that occurred relatively rarely (at most in 200 different articles) as links.

2. For each selected word, we collected all the links whose anchor texts consisted of that single word. We used every link as a training example: we created a feature-vector from each of these links.

3. For each selected word, we built a classifier, and performed cross-validation on the feature-vectors belonging to that word, trying to classify the senses of the current training examples. For the classification, we used linear support vector machines, as implemented in the Java version of the LIBLINEAR machine-learning library, because it proved to be fast and effective.

We experimented using various features. For the first experiment, we used Bag of Words representations as features by simply using the number of times the words appeared in the feature vector. We soon realized that we get better results if we use TF-IDF measure instead the simple frequencies.

We performed POS-tagging on the articles using the Stanford POS-tagger, and we used the POS-tag of the anchor text, the POS-tags of the surrounding words, the first nouns before and after the anchor, and the first verbs before and after the anchor as features in the examples. We used a bag of words of the four words left and the four words right of the anchor text (excluding stop words), and another Bag of Words of the at most five most frequent non stop-words occurring at least three times in the paragraph of the anchor text.

Using the features mentioned above, we measured 75.42% macro-averaged accuracy during a 5-fold cross validation repeated 5 times, on 100 randomly selected ambiguous words that occurred at least in 20, at most in 200 different articles.

We could improve our results to 86.80% macro-averaged accuracy on the same words by introducing the following features: a Bag of Words of the 100 non stop words to the left, and 100 non stop words to the right of the anchor, and another bag of words from all of the non stop words of the article that contains the link. This experiment indicates that the hierarchical structure of the feature-vector, using both narrower and wider contexts of the word is beneficial.

We also run the experiment on the 30 ambiguous nouns that were used in [45]. The author reached 84.65% on these words by mapping each Wikipedia-sense to a WordNet sense, and performing cross-validation on the resulting examples. Note that this mapping makes the task easier, because it reduces the number of senses. For example, the articles Christian Church and Catholic Church were both mapped to the WordNet sense {church, Christian church}. Since each Wikipedia-sense was mapped to exactly one WordNet sense, the number of senses could only be decreased.

We did not perform this mapping, mainly because it requires human work, and we want a fully automatic system. On the same ambiguous nouns, we reached 83.31%, which (despite the harder task) is very close to the results described in [45]. This is possible because of two reasons:

1. we used better features, which gather information from the whole article,

not just one paragraph

2. we had more than twice as many links, because since 2007, the Wikipedia grew bigger.

Note that the aforementioned experiments were run only on the *first-order* links of the anchor. Anchor texts of second-order links (i.e., the links that point to the same articles as the first-order links, but with different anchor texts) correspond to the same sense of the original anchor texts, so these second order anchor texts are additional examples with additional contexts. Therefore, one can use both first-order and second-order links in the *training* phase.

Measuring the accuracy of a classifier, which is trained on both *first-order* and *second-order* links gave surprising results. We found that there are cases when these extra training examples improve the results considerably, but sometimes the second-order links ruin the accuracy of the classifier. We need to investigate this issue further to see if in different topics the same meaning is used in different ways and contexts. These differences would distort the classifier.

Based on the topics learnt by our matrix factorization method, disambiguation filtered by topic may be possible. We plan to investigate this possibility further.

7.2.3 Extracting text

In the previous semesters, we have processed HTML files only to the extent required for the support vector machine in our crawler. We had to exclude most of the HTML tags, javascript, menu items, but we did not have to retain the order of the words, the complete sentences, etc., as we were working with a Bag of Words representation. This is not sufficient for many tasks, for example, parsing.

In this semester, we have developed a system to handle extraction of textual content from HTML files. Our first goal was flexibility: different tasks require different processing. For example, there are some applications where the comments on the page of a blog are important, in other applications they must be discarded. The system consists of independent filtering modules that can be combined in any way.

The processing of a HTML file consists of three steps. In the first step, we utilize `Apache TIKA` (`http://tika.apache.org/`) to separate the text from the non-textual content (e.g., HTML tags, javascript, etc.). In the second step, we process this text and generate a list of paragraphs. In the third step, the filters are run on this list.

The filters are run sequentially, the input of one filter is the output of the previous filter. Any number of filters can be combined this way to achieve the desired operation.

The filters can be used in 'discard mode', or in 'non-discard mode'. In discard mode, the paragraphs that are filtered out are removed from the list of paragraphs. In non-discard mode, only the content of the paragraphs is deleted,

the empty paragraph remains in the list. This is useful for example, when using the *ConnectedTextFilter*.

Here are the filters and the reasoning behind their utilization:

- *CommentFilter*: Discards all paragraphs after a short paragraph that contains the substring `omment`. The letter 'c' is omitted intentionally as a simple way of ensuring case-insensivity in the first letter. There are many pages where the comments are introduced by new lines and the word `comment(s)`.

- *ConnectedTextFilter*: Discards the paragraphs without at least one non-empty paragraph above or below them. On many pages, the content of the page consists of blocks of paragraphs. If a paragraph has no non-empty neighbor paragraphs, it is probably an advertisement, menu item, or comment. This filter can also be used to discard one-paragraph comments, but retain comments with more than one paragraphs (that are more likely to contain useful information).

- *MostParagraphsTextFilter*: Selects the longest connected section (i.e., that contains the most paragraphs). The longest connected section (i.e., the most consecutive paragraphs on a page) is most likely to be the main article on the page.

- *NumberOfCharactersFilter*: Removes all the paragraphs with less than *threshold* characters. Very short paragraphs are usually menu items or advertisements.

- *NumberOfLettersFilter*: The same as above, but only letters count. This helps filter out longer textual elements that have very few letters (e.g., separators).

- *NumberOfNewlinesFilter*: This filter discards all paragraphs where the number of new lines exceeds a threshold. In paragraphs that contains e.g., advertisements, there are often many new lines. In a paragraph containing real content, there are usually only a few.

- *ParagraphFilter*: A base abstract class. Does not do any filtering, but offers an extension point for new filters.

- *PostedByFilter*: Discards all paragraphs after a short paragraph that contains the substring `osted by`. The letter 'p' is omitted intentionally as a simple way of ensuring case-insensivity in the first letter. On blogs, there is usually a short paragraphs that contains `posted by` closing each article.

- *ProportionOfLinkedCharactersFilter*: This filter discards all the paragraphs where the majority of the content is linked. Most of the text of advertisements, menu items and sidebar items are linked.

- *PunctuationFilter*: A basic filter that removes all the paragraphs that do not contain any punctuation (".", "!", or "?"). Many menu items and advertisements do not contain punctuation.

7.2.4 Attached software

We have attached three software components to this report:

1. The architecture for mining Wikipedia can be downloaded from `http://nipg.inf.elte.hu/USAF/WikiRetrieval.zip`.
2. Our updated topical crawler can be found at `http://nipg.inf.elte.hu/USAF/Crawler.zip`. This software is also necessary for the third component to work.
3. The software that we used to conduct the experiments for detecting the most influential blogs based on document content can be found at `http://nipg.inf.elte.hu/USAF/CELF.zip`. This component also contains the module that indexes the webpages downloaded by our crawler. Requires the second component to work.

Bibliography

[1] Eneko Agirre and Philip Glenny Edmonds, *Word sense disambiguation: Algorithms and applications*, Springer, 2006.

[2] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami, *Mining association rules between sets of items in large databases*, SIGMOD Rec. **22** (1993), no. 2, 207–216.

[3] Jordi Atserias, Hugo Zaragoza, Massimiliano Ciaramita, and Giuseppe Attardi, *Semantically annotated snapshot of the english wikipedia*, Proceedings of the Sixth International Language Resources and Evaluation (LREC'08) (Marrakech, Morocco) (European Language Resources Association (ELRA), ed.), 2008.

[4] Sushil Bikhchandani, David Hirshleifer, and Ivo Welch, *A theory of fads, fashion, custom, and cultural change as informational cascades*, The Journal of Political Economy **100** (1992), no. 5, 992–1026.

[5] BNC Consortium, *The British National Corpus, version 2 (BNC World)*, 2001.

[6] Karsten M. Borgwardt, *Graph kernels*, Computer science, Ludwig-Maximilians-University, 2007.

[7] Ulrik Brandes, Marco Gaertler, and Dorothea Wagner, *Engineering graph clustering : Models and experimental evaluation*, ACM Journal of Experimental Algorithmics (2007).

[8] E. J. Candès, X. Li, Y. Ma, and J. Wright, *Robust principal component analysis?*, http://arxiv.org/abs/0912.3599, 2009.

[9] Soumen Chakrabarti, Martin van den Berg, and Byron Dom, *Focused crawling: a new approach to topic-specific web resource discovery*, 1999.

[10] Chih-Chung Chang and Chih-Jen Lin, *LIBSVM: a library for support vector machines*, 2001.

[11] Jordi Conesa, Veda C. Storey, and Vijayan Sugumaran, *Experiences using the researchcyc upper level ontology*, Lecture Notes in Computer Science, vol. Volume 4592/2007, pp. 143–155, Springer Berlin / Heidelberg, 2007.

[12] D. A. Cruse, *Polysemy: Theoretical and computational approaches*, ch. Aspects of the Micro-structure of Word Meanings, pp. 30–51, Oxford University Press, 2002.

[13] James R. Curran and Marc Moens, *Improvements in automatic thesaurus extraction*, Proceedings of the ACL-02 workshop on Unsupervised lexical acquisition (Morristown, NJ, USA), Association for Computational Linguistics, 2002, pp. 59–66.

[14] Jon Curtis, John Cabral, and David Baxter, *On the application of the cyc ontology to word sense disambiguation*, (2006).

[15] Wei Dai and Olgica Milenkovic, *Subspace pursuit for compressive sensing signal reconstruction*, IEEE Trans. Inf. Theor. **55** (2009), 2230–2249.

[16] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin, *Liblinear: A library for large linear classification*, J. Mach. Learn. Res. **9** (2008), 1871–1874.

[17] Illes J. Farkas, Daniel Abel, Gergely Palla, and Tamas Vicsek, *Weighted network modules*, New Journal of Physics **9** (2007), 180.

[18] M. L. Fisher, G. L. Nemhauser, and L. A. Wolsey, *An analysis of approximations for maximizing submodular set functions.*, Math. Programming Stud. (1978), no. 14. MR MR510369

[19] B. Gábor, V. Gyenes, and A. Lőrincz, *A corpus-based neural net method for explaining unknown words by WordNet senses*, Lecture Notes in Artificial Intelligence **3721** (2005), 470–477.

[20] Bálint Gábor, Viktor Gyenes, and András Lörincz, *Corpus-based neural network method for explaining unknown words by wordnet senses*, PKDD, 2005, pp. 470–477.

[21] Evgeniy Gabrilovich and Shaul Markovitch, *Computing Semantic Relatedness using Wikipedia-based Explicit Semantic Analysis*, IJCAI, 2007, pp. 1606–1611.

[22] Evgeniy Gabrilovich and Shaul Markovitch, *Wikipedia-based semantic interpretation for natural language processing.*, J. Artif. Intell. Res. (JAIR) **34** (2009), 443–498.

[23] John S. Garofolo, Cedric G. P. Auzanne, and Ellen M. Voorhees, *The TREC Spoken Document Retrieval Track: A Success Story*, RIAO, 2000, pp. 1–20.

[24] Jim Giles, *Internet encyclopaedias go head to head*, Nature **438** (2005), no. 7070, 900–901.

[25] Eui-Hong Han and George Karypis, *Centroid-based document classification: Analysis and experimental results*, PKDD, LNCS, vol. 1910, 2000, pp. 116–123.

[26] Zellig Harris, *Distributional structure*, Word **10** (1954), no. 23, 146–162.

[27] Cho-Jui Hsieh, Kai-Wei Chang, Chih-Jen Lin, S. Sathiya Keerthi, and S. Sundararajan, *A dual coordinate descent method for large-scale linear svm*, ICML '08: Proceedings of the 25th international conference on Machine learning (New York, NY, USA), ACM, 2008, pp. 408–415.

[28] C. W. Hsu and C. J. Lin, *A comparison of methods for multiclass support vector machines*, IEEE Transactions on Neural Networks **13** (2002), no. 2, 415–425.

[29] Witten I.H., Paynter G.W., Frank E., Gutwin C., and Nevill-Manning C.G., *Kea: Practical automatic keyphrase extraction*, DL '99, 1999, (Poster presentation.), pp. 254–256.

[30] R. Jenatton, J. Mairal, G. Obozinski, and F. Bach, *Proximal methods for sparse hierarchical dictionary learning*, ICML, 2010.

[31] Rodolphe Jenatton, Guillaume Obozinski, and Francis Bach, *Structured sparse principal component analysis*, AISTATS, 2010.

[32] Thorsten Joachims, Fachbereich Informatik, Fachbereich Informatik, Fachbereich Informatik, Fachbereich Informatik, and Lehrstuhl Viii, *Text categorization with support vector machines: Learning with many relevant features*, 1997.

[33] Paul B. Kantor and Ellen M. Voorhees, *The TREC-5 Confusion Track: Comparing Retrieval Methods for Scanned Text*, Information Retrieval **2** (2000), 165–176.

[34] David Kempe, Jon Kleinberg, and Éva Tardos, *Maximizing the spread of influence through a social network*, KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining (New York, NY, USA), ACM, 2003, pp. 137–146.

[35] Samir Khuller, Anna Moss, and Joseph (Seffi) Naor, *The budgeted maximum coverage problem*, Inf. Process. Lett. **70** (1999), no. 1, 39–45.

[36] Pranam Kolari, Tim Finin, and Anupam Joshi, *SVMs for the Blogosphere: Blog Identification and Splog Detection*, AAAI Spring Symposium on Computational Approaches to Analysing Weblogs, Computer Science and Electrical Engineering, University of Maryland, Baltimore County, March 2006, Also available as technical report TR-CS-05-13.

[37] J. Kruskal, *Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis*, Psychometrika **29** (1964), 1–27.

[38] Amy N. Langville and Carl D. Meyer, *Deeper inside pagerank*, Internet Mathematics **1** (2004), no. 3, 335–380.

[39] Claudia Leacock, George A. Miller, and Martin Chodorow, *Using corpus statistics and wordnet relations for sense identification*, Comput. Linguist. **24** (1998), no. 1, 147–165.

[40] Yoong Keok Lee and Hwee Tou Ng, *An empirical evaluation of knowledge sources and learning algorithms for word sense disambiguation*, EMNLP, 2002, pp. 41–48.

[41] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance, *Cost-effective outbreak detection in networks*, KDD '07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining (New York, NY, USA), ACM, 2007, pp. 420–429.

[42] Dekang Lin, *Automatic retrieval and clustering of similar words*, Proceedings of the 17th international conference on Computational linguistics (Morristown, NJ, USA), Association for Computational Linguistics, 1998, pp. 768–774.

[43] Julien Marial, Francis Bach, Jean Ponce, and Guillermo Sapiro, *Online learning for matrix factorization and sparse coding*, Journal of Machine Learning Research **11** (2010), 10–60, `http://www.jmlr.org/papers/volume11/mairal10a/mairal10a.pdf`.

[44] Olena Medelyan, David Milne, Catherine Legg, and Ian H. Witten, *Mining meaning from Wikipedia*, International Journal of Human-Computer Studies **67** (2009), 716 – 754.

[45] Rada Mihalcea, *Using Wikipedia for automatic word sense disambiguation*, NAACL, 2007, pp. 196–203.

[46] George A. Miller, *WordNet: A lexical database for English*, Communications of the ACM **38** (1995), 39–41.

[47] Gordon Mohr, Michael Stack, Igor Ranitovic, Dan Avery, and Michele Kimpton, *An introduction to heritrix*, 4th International Web Archiving Workshop, 2004.

[48] Roberto Navigli, *Word sense disambiguation: A survey*, ACM Comput. Surv. **41** (2009), 1–69.

[49] J. Nivre, J. Hall, and J. Nilsson, *Maltparser: A data-driven parser-generator for dependency parsing*, Proceedings of the fifth international conference on Language Resources and Evaluation (LREC) (Genoa, Italy), 2006, pp. 2216–2219.

[50] Erkki Oja, *Neural networks, principal components, and subspaces*, Int. J. Neural Syst. **1** (1989), no. 1, 61–68.

[51] Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek, *Uncovering the overlapping community structure of complex networks in nature and society*, Nature **435**, no. 7043, 814–818.

[52] T. Pedersen, S. Patwardhan, and J. Michelizzi, *Wordnet::similarity - measuring the relatedness of concepts*, 2004.

[53] D. T. Pham and M. Congedo, *Least square joint diagonalization of matrices under an intrinsic scale constraint*, Independent Component Analysis and Signal Separation (T. Adali et al., ed.), Lecture Notes in Computer Science, vol. LNCS 5441, Springer, 2009, pp. 298–305.

[54] M. F. Porter, *An algorithm for suffix stripping*, (1997), 313–316.

[55] Satu Elisa Schaeffer, *Graph clustering*, Computer Science Review **1** (2007), 27 – 64.

[56] Rion Snow, Sushant Prakash, Dan Jurafsky, and Andrew Y. Ng, *Learning to merge word senses*, Proceedings of EMNLP, 2007, `http://ai.stanford.edu/~ang/papers/emnlp07-senseclustering.pdf`.

[57] F. J. Theis, T. P. Cason, and P. A. Absil, *Soft dimension reduction for ica by joint diagonalization on the stiefel manifold*, Independent Component Analysis and Signal Separation (T. Adali et al., ed.), Lecture Notes in Computer Science, vol. LNCS 5441, Springer, 2009, pp. 354–361.

[58] P. Tichavsky, A. Yeredot, and J. Nielsen, *A fast approximate joint diagonalization algorithm using a criterion with a block diagonal weight matrix*, IEEE International Conference on Acoustics, Speech and Signal Processing, 2008, pp. 3321–3324.

[59] L.J.P. van der Maaten and G.E. Hinton, *Visualizing high-dimensional data using t-sne*, (2008).

[60] Vladimir N. Vapnik, *The nature of statistical learning theory (information science and statistics)*, Springer, November 1999.

[61] S. V. N. Vishwanathan, K. M. Borgwardt, I. R. Kondor, and N. N. Schraudolph, *Graph kernels*, arXiv.org > cs > 0807.0093, July 2008, submitted to the Journal of Machine Learning Research.

[62] Wikipedia, *Wikipedia*, `http://en.wikipedia.org/wiki/Wikipedia`.

[63] John Wright, Arvind Ganesh, Shankar Rao, and Yi Ma, *Robust principal component analysis: Exact recovery of corrupted low-rank matrices via convex optimization*, Conference on Neural Information Processing Systems (NIPS 2009), December 2009.

[64] John Wright, Allen Y. Yang, Arvind Ganesh S. Shankar Sastry, and Yi Ma, *Robust face recognition via sparse representation*, PAMI, 2008.

[65] Justin Zobel and Alistair Moffat, *Exploring the similarity space*, SIGIR Forum **32** (1998), 18–34.